University of Alberta

IMPLEMENTATION OF JPEG IMAGE COMPRESSION ON THE DSP-RAM SINGLE-CHIP SIMD ARCHITECTURE

by

Hua Ai

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of

the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87706-X
Our file Notre référence
ISBN: 0-612-87706-X

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Hua Ai

Title of Thesis: Implementation of JPEG Image Compression on the DSP-RAM Single-Chip SIMD Architecture

Degree: Master of Science

Year this Degree Granted: *2003*

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

_____

Signature

:ring

Date : Sept. 23, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Implementation of JPEG Image Compression on the DSP-RAM Single-Chip SIMD Architecture** submitted by **Hua Ai** in partial fulfillment of the requirements for the degree of **Master of Science**.

Date: Sept. 19, 2003

ABSTRACT

DSP-RAM is a programmable and scalable SIMD (Single Instruction stream, Multiple Data stream) architecture that integrates multiple processor-memory pairs on a single chip in order to deliver high-performance computational power for multimedia and communications applications at a relatively low clock frequency. This thesis describes research that investigated the advantages and limitations of DSP-RAM when dealing with important computationally-intensive functions in image processing and digital signal processing.

The main objective of this research project was to investigate and improve the DSP-RAM architecture's ability to deal with system-level problems. We examined DSP-RAM's functional versatility, analyzed the pipeline structure and behavior of DSP-RAM, and summarized the main factors that affect the efficiency of DSP-RAM implementations. We also developed a method to improve the programmability of DSP-RAM.

Then we used still image JPEG encoding and the Discrete Wavelet Transform (DWT) as benchmark problems. The performance of the DSP-RAM implementation of the JPEG encoder is evaluated and analyzed against the well-known IJG JPEG software encoder executed on a 1.6-GHz Pentium 4 machine.

We also implemented an optimized and balanced DSP-RAM architecture with the latest FPGA technology. An evaluation of the resource requirements and the achievable performance of the FPGA implementation were also performed.

To my wife Hanbin Zhang and our families.

ACKNOWLEDGMENTS

I would like to thank Dr. Bruce F. Cockburn for his supervision and help. Without his timely guidance, inspiration and encouragement, this thesis could not have been done. I want to especially thank him for taking the responsibility of being my supervisor in the summer of 2001. Without his trust, I would not be doing research in my favorite area.

I would also like to thank Micronet R&D and the Natural Sciences and Engineering Research Council of Canada for their financial support.

My final thanks go to my wife Hanbin Zhang and our families. They provided me endless support and tolerated my neglect, they were always the source of my power and courage.

TABLE OF CONTENTS

Chapter 1 Introduction *1*

1.1 Background on Microprocessor Architecture Design 1

1.2 DSP-RAM Research 5

1.3 DSP-RAM for Mobile Multimedia Computing 6

1.3.1 Requirements for Future Mobile Processors 6

1.3.2 Multimedia-centric Applications and the DSP-RAM Architecture 7

1.4 Thesis Topics 9

1.5 General Approach Taken in the Thesis 10

1.6 Thesis Organization 11

Chapter 2 Background *12*

2.1 Reviews of the JPEG and JPEG 2000 Standards 12

2.1.1 Background 12

2.1.2 Architecture of the DCT-based JPEG Encoder 16

2.1.3 The Discrete Wavelet Transform in JPEG 2000 27

2.2 Review of DSP-RAM 30

2.2.1 The DSP-RAM Architecture 30

2.2.2 Past Research 34

Chapter 3 Improved JPEG and DWT Implementations *37*

3.1 System-level Challenges for the DSP-RAM Architecture 37

3.1.1 DSP-RAM's Functional Versatility 37

3.1.2 Factors Affecting Overall Performance 38

3.1.3 DSP-RAM's Programmability 48

3.2 JPEG Implementation on DSP-RAM 56

3.2.1 Data Initialization and Organization 57

3.2.2 Processing Pattern 58

3.2.3 Implementation of Color Conversion 60

3.2.4 Implementation of Subsampling 62

3.2.5 Implementation of DCT 64

3.2.6 Implementation of Quantization 66

3.2.7 Entropy Encoder and File Format Processing 67

3.3 DWT Implementation on DSP-RAM 68

3.3.1 DWT Processing in JPEG 2000 68

3.3.2 Data Flow of the DSP-RAM Implementation of DWT 71

3.3.3 Processing Flow of the DSP-RAM Implementation of DWT 73

3.3.4 Clock Cycles for the DSP-RAM Implementation of DWT 74

Chapter 4 FPGA Implementation of the DSP-RAM Architecture *76*

4.1 Design Platform and Design Target 76

4.2 Detailed Implementation of DSP-RAM on FPGA 77

4.2.1. Top-level Architecture 77

4.2.2. Implementation of the RAM Component 79

4.2.3. Implementation of the ALU Component 80
4.2.4. Implementation of the Data Path Switch Component 80

4.3 Analysis of Performance and Resource Requirements 81

4.4 Conclusions 84

Chapter 5 Evaluation 86

5.1 Evaluation Platform and Methods 86
5.1.1. Evaluation Platform and Methods for DSP-RAM Implementations 86
5.1.2. Evaluation Platform and Methods for Sequential Implementations 87

5.2 Evaluation Results 90
5.2.1. Run-time Statistics for DSP-RAM Implementations 90
5.2.2. Run-time Statistics for Sequential Implementations 92

5.3 Analysis of Results 93
5.3.1. Analysis for Parallel Parts of Encoding 93
5.3.2. Analysis for Overall JPEG Encoding Process 94
5.3.3. Approaches to Further Improve the DSP-RAM JPEG encoder 97
5.3.4. Upgrade to Motion JEPG Encoder 104

Chapter 6 Conclusions 105

6.1 Summary 105

6.2 Future work 106

Bibliography 108

LIST OF TABLES

Table 2.1-1 Computational complexity of various fast DCT algorithms 23
Table 2.1-2 Computational complexity of scaled DCT algorithms 24
Table 3.1-1 List of operations supported by a typical DSP-RAM system 38
Table 3.1-2 Calculation on DSP-RAM without considering data hazards 45
Table 3.1-3 Calculation on DSP-RAM that avoids data hazards 45
Table 3.1-4 Pipeline control flow for the vector computation AX+BY+C 53
Table 3.3-1 Clock cycles needed for a PE to perform a DWT on a MCU 75
Table 4.1-1 Resources list for the Virtex-II Pro Platform FPGAs 77
Table 4.3-1 Performance and resource requirements for major components 81
Table 4.3-2 Performance and resources for a single PE and groups of PEs 82
Table 4.3-3 Timing closure of an FPGA implementation for a 2×16 PE matrix 83
Table 5.1-1 Hardware and software platform for evaluating sequential algorithms 88
Table 5.1-2 Performance numbers obtained for the original version of the IJG library and for the modified version with the Intel Integrated Performance Primitives (IPP) 89
Table 5.2-1 Runtime statistics for the parallel parts of the DSP-RAM JPEG encoder 90
Table 5.2-2 Runtime distribution for parallel functions in the DSP-RAM JPEG encoder 92
Table 5.2-3 Runtime statistics for the IJG JPEG encoder 93
Table 5.3-1 Performance improvement for the parallel parts of encoding 94
Table 5.3-2 Speed-up for a JPEG encoder on a Pentium 4 system enhanced by a DSP-RAM co-processor 95
Table 5.3-3 Performance requirement for the host controller when a DSP-RAM competes with a Pentium 4 96
Table 5.3-4 True table for the detection logic 101

LIST OF FIGURES

Figure 2.1-1 Generic compression system 14
Figure 2.1-2 Tradeoffs in choosing compression methods 15
Figure 2.1-3 JPEG 2000 vs. JPEG 16
Figure 2.1-4 Block diagram of a JPEG encoder 16
Figure 2.1-5 Illustration of the downsampling process 19
Figure 2.1-6 64 8 × 8 DCT spatial basis functions 20
Figure 2.1-7 Using the 1-D DCT to compute the 2-D DCT 22
Figure 2.1-8 A typical coefficients matrix after quantization 25
Figure 2.1-9 Zigzag ordering in an 8×8 matrix 26
Figure 2.1-10 Sub-band coding decomposition of DWT 29
Figure 2.2-1 DSP-RAM's top-level architecture 31
Figure 2.2-2 Diagram of DSP-RAM's PE architecture 32
Figure 2.2-3 Detailed PE architecture 33
Figure 2.2-4 Billie Kwan's parallel 2-D DCT algorithm on DSP-RAM 35
Figure 3.1-1 Simplified DSP-RAM pipeline 43
Figure 3.1-2 Comparison between DSP-RAM's pipeline and a black-box pipeline 50
Figure 3.1-3 Modified DSP-RAM black-box pipeline 51
Figure 3.1-4 Control State Machine for vector computation AX+BY+C 54
Figure 3.1-5 Input state machine for the vector computation AX+BY+C 55
Figure 3.1-6 Output state machine for the vector computation AX+BY+C 55
Figure 3.2-1 Block diagram of our targeted JPEG encoder 56
Figure 3.2-2 Data flow in the DSP-RAM implementation 60
Figure 3.2-3 Comparison of clock cycles needed for DSP-RAM and a non-parallel architecture to perform color conversion function 62
Figure 3.3-1 Top diagram of multi-level DWT processing of an 8×8 pixel block 69
Figure 3.3-2 Using a 1-D DWT to perform a 2-D DWT transformation 69
Figure 3.3-3 Use of a reversible transformation to construct 1-D DWT filters 70
Figure 3.3-4 12 patterns of data flows in the DSP-RAM implementation of DWT 72
Figure 3.3-5 State transition diagram for the control of DWT on DSP-RAM 74
Figure 4.2-1 Top-level architecture of the FPGA implementation of DSP-RAM 78
Figure 4.2-2 Top-level architecture of the FPGA implementation of a PE 79
Figure 4.2-3 Architecture of the data path switch component 81
Figure 5.2-1 How the number of PEs affects the performance of the DSP-RAM JPEG encoder 91
Figure 5.3-1 The DSP-RAM implementation of the differential coding 99
Figure 5.3-2 Modified branch control circuit for run-length coding 100
Figure 5.3-3 Parallelized Huffman coding on the current DSP-RAM architecture 101
Figure 5.3-4 Using CAM to simplify the table-lookup operation 102
Figure 5.3-5 Adding an address register to realize table lookup in DSP-RAM 103
Figure 5.3-6 Data flow of a proposed DSP-RAM MJPEG encoder 104

GLOSSARY

1-D One-dimensional.

2-D Two-dimensional.

ALU Arithmetic and Logic Unit.

AMD Advanced Micro Devices, Inc.

bpp bits per pixel.

BMP Microsoft Windows bitmap format.

C-RAM Computational RAM.

CAM Content-Addressable Memory.

CCIR Consultative Committee for International Radio.

CPI Clock Cycles per Instruction.

CPU Central Processing Unit.

DCT Discrete Cosine Transform.

DSP Digital Signal Processing or Digital Signal Processor.

DWT Discrete Wavelet Transform.

FDWT Forward Discrete Wavelet Transform.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

fps frames per second.

HVS Human Visual System.

IJG Independent JPEG Group.

ILP Instruction Level Parallelism.

IPP Integrated Performance Primitives.

IRAM Intelligent Random-Access Memory.

ISO International Organization for Standardization.

ITU International Telecommunication Union.

JFIF JPEG File Interchange Format.

JPEG Joint Photographic Experts Group.

MAC Multiply-Accumulate or Multiplier-Accumulator.

MAX MAX-1 and MAX-2 are small sets of multimedia acceleration extensions implemented in PA-RISC processors from Hewlett-Packard.

MCU Minimum Coded Unit.

MMX a set of multimedia extensions for the Intel Pentium architecture.

MPEG Moving Pictures Expert Group.

PE Processing Element.

PGM Portable Grey Map file format.

PLP Process-Level Parallelism.

PPM Portable Pixmap.

SIMD Single Instruction stream, Multiple Data stream.

SISD Single Instruction stream, Single Data stream.

SPARC Scalable Processor ARChitecture, from Sun Microsystems.

SSE Streaming SIMD Extensions, for the Intel Pentium processors.

TIFF Tagged Image File Format.

TLP Thread-Level Parallelism.

TGA Truevision Targa Graphics Adaptor file format.

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.

Chapter 1

INTRODUCTION

Over the past 20 years, mainstream microprocessor and computer architectures have been continuously adapted to meet the demands of dealing with mostly non-real-time and stand-alone applications. The recent surge in demand for multimedia and mobile applications has drawn more attention to parallel computing architectures. Research shows that the SIMD (Single Instruction stream, Multiple Data stream) style and tightly coupled memory-processor architectures, like Computational RAM and DSP-RAM, could be strong competitors in the mobile multimedia computing domain because of their advantages in energy efficiency, data parallelism and compact system size, etc. [1,2,3,4,5,6,7] At the University of Alberta, researchers have investigated implementations on the proposed DSP-RAM architecture of key individual functional blocks of the JPEG still image and MPEG video compression standards, i.e. the DCT (Discrete Cosine Transform) and vector quantization [8,9]. However, an investigation of the entire problem, which involves system-level optimizations of power, cost, performance, and some architectural issues such as data reorganization and data flow control, has until now not yet been done.

The research described in this thesis focused on investigating the suitability of DSP-RAM for implementing the JPEG and JPEG-2000 still image compression. Modifications and optimizations to the original DSP-RAM architecture were then proposed to improve its programmability and efficiency.

1.1 Background on Microprocessor Architecture Design

Advances in integrated circuit technology will soon allow the integration of one billion transistors on a single chip. A challenge for computer architects and designers will be to propose microprocessor designs that use this huge transistor budget efficiently and meet the requirements of future applications.

Nearly all modern microprocessors, including the Pentium, PowerPC, Alpha, and SPARC microprocessors, are superscalar. They integrate multiple functional units into

their datapath pipelines and allow multiple instructions to be executed per clock cycle. However, because of their relatively poor scalability and modularity, the costs of design and verification increase exponentially with increases in system size [10]. Consequently, in order to use the available transistor budget efficiently, the computer design community will have to develop more powerful tools or methodologies to reduce the costs of design and verification and/or make their architectures more scalable.

Various ways are being investigated to face the forthcoming challenges and opportunities. Some of the design strategies could be usefully combined, such as parallel processing, processor-memory integration, and scalable architecture.

Some of these ways, such as advanced superscalar architectures, superspeculative instruction execution, simultaneous multithreading, trace processors, and chip multiprocessors [10,11,12,13,14], are based on, or partially based on, current superscalar architectures. The main differences between them are the level where the parallelism is applied and the core complexity. Advanced superscalar and superspeculative architectures focus on further exploiting Instruction Level Parallelism (ILP). Simultaneous multithreading architectures improve the utilization of functional units by exploiting parallelism at both the instruction level and the thread/process level. Trace processors dynamically detect cache traces (sequences of instructions), consider them as basic units for execution, optimize them and process them on different cores in a parallel or speculative manner. Single-chip multiprocessor architectures have multiple simple cores and mainly exploit parallelism at the process level. These architectures inherit and extend many features from superscalar architectures, but they also inherit some weak points of current microprocessor architectures, for instance, the inefficiency of exploiting the data parallelism inherent in many multimedia applications.

The rising demand of multimedia applications today is consuming more and more clock cycles of computers. The performance of multimedia applications hence becomes a very important factor in processor design. Major vendors of general-purpose processors have incorporated SIMD features into their processor architectures and have announced SIMD extensions to their instruction sets to enhance the support of multimedia applications. Examples include the MAX-2 extensions to Hewlett-Packard's PA-RISC; the MMX, SSE, and SSE-2 for Intel's x86; the MDMX extensions to the

MIPS V; and AMD's 3D-Now! technology, etc. [15] These technologies extended the original scalar architectures while preserving backward compatibility and ensured the modifications were transparent to operating systems and application software.

To further exploit the data parallelism of multimedia applications with SIMD architectures, a natural approach is to increase the number of processing units (PU) in the architecture. This approach has not been effective until recently because of the limitations of integrated circuit technology. For those former SIMD systems, which implemented PE and memory on different chips, or even on different printed circuit boards, increasing the number of PEs will greatly increase system size and power consumption, and decrease the maximum working frequency of the system. With today's integrated circuit technology, however, it's possible to solve these problems by integrating the main parts of the system, i.e., PE, communication network, memory, and even the instruction dispatcher (host controller), into one single chip.

The .maturity of these conditions has raised interest in SIMD-style, processor-in-memory architectures over the past 5 years. UC-Berkeley's IRAM [16], the University of Alberta and University of Toronto's Computational RAM [1,2], and the University of Alberta's DSP-RAM [6,7,8,9] are among these architectures.

Compared to traditional processor architectures, memory bandwidth is typically an important concern for SIMD architectures. To process more data at a time means we need to have more data available to process at a time, which leads to a greater demand on memory bandwidth. SIMD architectures like C-RAM, DSP-RAM and IRAM improve the memory system bandwidth greatly by integrating processors and memory in one single chip. Because on-chip wire capacitances are much smaller than the capacitances on pins and off-chip wires, an on-chip data bus can be operated at a much higher frequency and at lower power. Without the limitation of the relatively small number of pins on a die, the width of the internal data bus can also be increased greatly.

Although IRAM, C-RAM and DSP-RAM are all SIMD architectures, the organization and operation of their PEs is different. IRAM can be categorized as a Vector architecture [17,18], with a relatively small number of powerful PEs performing operations on its vector register file, and sharing the access to memory through a

crossbar network. C-RAM and DSP-RAM should be categorized as processor array architectures, which have a large number of PEs, each of which works on small chunks of data (bit, byte, word, etc.) in its own local memory and communicates through a one-dimensional shift-register network. Unlike IRAM and C-RAM, the current version of DSP-RAM adopts a memory-register architecture for its PE, which simplifies the implementation but limits the clock frequency of its ALU according to the speed of its memory access.

Among these three architectures, the C-RAM architecture can achieve the greatest memory bandwidth and the most massive parallelism because it pitch-matches its super-simplified bit-serial PE (Processing Element) directly to the sense amplifier of the memory array. However, because of the simplicity of the PE architecture, the computational power of each PE is relatively weak. For example, to perform an 8-bit addition on the enhanced C-RAM architecture [1], which pitch-matches its PE to four sense amplifiers, it needs four DRAM memory accesses and eight bit-serial additions in total. Research shows that, for those applications with moderate internal parallelism, such as DCT (Discrete Cosine Transform) or motion estimation, which require computation on a medium amount of data, C-RAM's massive parallelism might not be fully utilized, and the overall performance might be disappointing because of the small computational power of each PE.

The DSP-RAM architecture is a new attempt to utilize the power brought by the combination of SIMD architecture and memory-processor integration and is still at its very early stages of investigation. It could be considered as a variation of the C-RAM architecture. The current version of DSP-RAM, which is mainly aimed at applications with moderate levels of data parallelism (i.e. < 1000), enhances the computational power of its PE by adopting a pipelined multiple-bit MAC as its ALU. It also incorporates a one-dimensional shift register network and a broadcast bus to enhance the communication among its PEs and the host controller. Unlike IRAM and C-RAM, the current version of DSP-RAM uses SRAM instead of DRAM as its on-chip memory for faster memory access and simpler potential implementation.

Based on the fact that general-purpose microprocessors are highly sophisticated for dealing with scalar computations but relatively awkward at exploiting the data

parallelism of vector computations, it's possible that the future billion transistor processors will have highly-scalable on-chip SIMD subsystems working seamlessly with one or more general-purpose cores to improve the performance of suitable applications or functions.

1.2 DSP-RAM Research

In 1999, the first DSP-RAM architecture was proposed at the University of Alberta [19]. It combined the concepts of DSP, SIMD and processor-in-memory into one co-processor architecture. A simulation prototype of this architecture was built in C++.

In 2001, a VHDL simulator for DSP-RAM was built in University of Alberta and its Instruction Set Architecture (ISA) was drafted. Researchers began to evaluate this architecture by implementing various algorithms on it. Simulation results showed that this architecture had great potential on some applications with moderate data parallelism, such as DCT, Wavelet transform, Vector Quantization, FIR (Finite Impulse Response) filtering, etc. [7,8]

In 2002, a VHDL implementation of DSP-RAM was synthesized on Xilinx Vertex FPGA devices. More functions in multimedia and wireless communication were evaluated on the DSP-RAM architecture [9]. Minor modifications were proposed to the original architecture to meet the requirements of the new applications. The area requirements and the efficiency of internal components of PE were also analyzed.

Until now, all the evaluations and improvements of DSP-RAM have been motivated by only a few specific functions. To solve a system-level problem, functions have to be scheduled, integrated and probably modified according to the circumstances. Data structures have to be generated, maintained, and updated. During these procedures, many issues, such as optimizing overall performance, data reorganization, memory management, system scalability, hardware utilization, need to be properly investigated for DSP-RAM. It's possible that an architecture optimized for different specific functions may not optimized for a system-level problem that incorporates these functions. For example, as we will discuss in detail in Section 2.2.2, Billie Kwan's DSP-RAM implementation of the 2-D DCT algorithm achieved high computing

performance with one DCT coefficient per clock cycle, it is not an optimized way to be integrated with other functions of JPEG encoding.

Thus, research on further evaluating and improving the ability of the DSP-RAM to solve system-level problems should now be a priority.

1.3 DSP-RAM for Mobile Multimedia Computing

With the development of information technology over the past 20 years, people are already able to send and receive considerable volumes of information to handheld devices such as cell phones and networked PDA. Now the problem is that people find that they do not have the enough computational power to process the information in the same way as they can do on a desktop, especially for computationally-intensive applications such as multimedia applications. This is because the microprocessor architectures that they are using in their handheld devices were originally designed for desktop/server applications, where scalar operations dominate and power consumption budget is not as tight as in a mobile computing environment.

In the following subsections, we will briefly review the requirements for future mobile multimedia processors and analyze the potential role that DSP-RAM could play.

1.3.1 Requirements for Future Mobile Processors

Future mobile processors have to meet the following requirements [15,20]:

High Performance for Multimedia Functions

Multimedia applications are very important for the usability, quality, productivity and enjoyment of future mobile computers, thus the ability to process high-performance multimedia functions will become very important for future mobile processor.

Energy Efficiency

In the mobile computing domain, energy consumption is a limiting factor for processor design. Increasingly, weight and battery life are more important than pure processing speed. Generally, the power budget for the whole device is much less than two watts [20]. A mobile processor should consume no more than one watt [20]. This requirement disqualifies those high-performance microprocessors with power budgets of tens of watts (e.g. Pentium III and 4).

Small Size

The future mobile computers are expected to be small enough to put into a pocket and weigh typically 6-12 ounces. This requirement implies innovations in the processor/computer architectures. Single-function chips like memories, coprocessors and hardware codecs will likely be integrated into the processor chips.

Low Design Complexity

Design complexity, along with scalability, is a common concern for processor design. Although microprocessor architects will soon have a billion-transistor budget for their designs, the future architecture might best be modularized and made scalable because of the limitations on verification costs and signal propagation effects over long wires. For future processor designs, long wires are expected to be avoided because the clock frequency will be so high that signals can only be driven within a small region (i.e. timing domain) during a single clock cycle.

1.3.2 Multimedia-centric Applications and the DSP-RAM Architecture

Current microprocessor architectures are mostly designed for and driven by general-purpose desktop/server applications and computer architectures. They are sophisticated at dealing with complex scheduling, processing, interrupts, etc. The rising demands of multimedia applications in the past 10-15 years have driven the computer industry to develop many new technologies to make current computer systems better meet the requirements of multimedia-centric applications. However, these technologies are still mostly based on the mainstream computer and microprocessor architectures, many of them do not apply in mobile multimedia computing domain [15,20,21].

What then are the characteristics of multimedia-centric applications and why could DSP-RAM architecture be a better competitor in this domain?

Real-time Response

Real-time response is essential for many media applications, which implies the requirement for great computational power. However, the superscalar architecture of current general-purpose processors limits their ability to exploit the data parallelism of multimedia applications. Generally, they increase their computational power by increasing the clock frequency and/or by using additional expensive dedicated

hardware, which tends to increase the power consumption and/or the system size and are therefore not desirable in mobile computing domain. The DSP-RAM architecture adopts a SIMD architecture to exploit the inherent data parallelism of multimedia-centric applications so that the system can be run at a relatively low frequency. Its scalable architecture enables us to enhance its computational power by simply (and with little additional design cost) increasing the number of processing elements.

Continuous-media Data Types

Media applications typically involve processing a continuous stream of data. This implies that temporary locality in data memory accesses is no longer a valid assumption. Temporary locality is assumed by current computer architectures and is the reason why large memory systems are needed and why the multi-level cache system works. For the processing of continuous-media data types, large memory systems are not necessary because the same multimedia information does not need to be stored and reloaded frequently. Also, a conventional cache system will be counter-productive here because it will experience cache misses continuously as new data arrives to be processed.

Fine-grained Parallelism

Most of the multimedia functions require performing the same sequences of operations on the data stream in a SIMD (Single-Instruction Stream, Multiple-Data Stream) fashion. Current superscalar processors tend to exploit Instruction-Level Parallelism (ILP) with advanced pipeline technologies, speculation and branch prediction [22]. Simultaneous multithreading tends to exploit Thread-Level Parallelism (TLP) and ILP [10,12]. Multiprocessors tend to exploit Process-Level Parallelism (PLP) [1014]. However, the inherent data parallelism of multimedia applications is different from these kinds of parallelism. Thus, the conventional architectures and its successors can not exploit it efficiently; however, DSP-RAM is optimized precisely to exploit this kind of data parallelism.

Coarse-grained Parallelism

In many multimedia applications, a pipeline of functions processes a stream of data to obtain the end results; these functions can often operate in parallel. Furthermore, it may become common for a mobile multimedia processor to deal with several multimedia

applications simultaneously. For example, a user may want to write a paper with voice recognition software while he/she is listening to music.

High Instruction Reference Locality

Unlike the data flow, the instruction flow of multimedia applications generally has high temporal and spatial locality. The operations within a few loops or function kernels tend to dominate the processing time. So, although the number of operations could be huge, the code size may be relatively small, the branches are predictable and loops are relatively easy to be unrolled to be processed in parallel. This feature makes the SIMD style of architecture a potentially better candidate for multimedia applications.

High Memory Bandwidth

Many multimedia applications require a huge memory bandwidth for large data sets that have limited locality. In another words, large capacity may be not required for the memory system, but large average data bandwidth is typically required. Current general-purpose superscalar processors generally have multiple function units integrated in their pipeline architecture, feeding data through a single data bus from relatively slow memory [22,23,24,25]. The memory bandwidth becomes a major problem of scalar architectures. Although most of the modern superscalar architectures employ a cache memory system to mitigate the bandwidth problem, it is ineffective in multimedia applications as we mentioned in the previous section: Continuous-media Data Types. The SIMD architecture combined with the processor-in-memory architecture, however, is probably a reasonable solution for this problem.

1.4 Thesis Topics

In this thesis, we are particularly interested in DSP-RAM's ability to solve system-level applications. There are many problems involved with this topic, for example, how does DSP-RAM work seamlessly with a host processor? What kinds of functions should be designated to DSP-RAM and what kinds of functions should be undertaken by the host? How to design the correct parallel algorithm for different functions to utilize the DSP-RAM architecture efficiently? How to reduce the context switching overhead between different functions? Are there any modifications to the DSP-RAM architecture that should be made to enhance its programmability and the data availability among the

internal components of its PE? How about the scalability of the system and the hardware utilization of the architecture?

It's important to evaluate DSP-RAM using system-level metrics. When we use DSP-RAM to deal with a specific algorithm, we only need to consider the complexity of the algorithm itself. But if we put the problem with a system perspective, we may have to be concerned with how the data is packed from the upstream functional blocks and how the data should be repacked to benefit the implementation of the downstream functional blocks. Therefore, an implementation optimized for an algorithm alone may not be efficient when it is put in a system perspective if it introduces too much overhead on data preparation.

1.5 General Approach Taken in the Thesis

To evaluate DSP-RAM's ability to operate efficiently at the system level, we chose the JPEG encoder as our main benchmark application.

JPEG is the most popular still image compression standard and is widely used in all kinds of applications [26]. It contains five main functional blocks in the whole process; some of them are also widely used in other image, video processing standards. Some of these functional blocks have already been implemented on DSP-RAM at University of Alberta. These earlier implementations can serve as references for the system-level implementation.

We analyzed the process flow and data flow of the sequential JPEG encoder, and then designed the process flow and data flow for parallel implementation. Then we developed sequential and parallel algorithms for those functional blocks, and embedded them into our process flow to finish the whole parallel implementation. During the procedure, we made modifications to the DSP-RAM architecture in order to meet the requirements of the system implementation. We also analyzed the effects on the overall performance of choosing different algorithms. Finally, we evaluated the performance of our final parallel implementation by comparing it to the published sequential implementation of JPEG encoder from the IJG (Independent JPEG Group). To further evaluate DSP-RAM's ability in data management and reorganization, we also implemented JPEG 2000's reversible FDWT (Forward Discrete Wavelet Transform)

on DSP-RAM. To obtain the hardware information required by our evaluation, we optimized the DSP-RAM architecture, combined it with state-of-the-art FPGA technology, and implemented it on Xilinx Virtex II Pro FPGA devices. The resource requirements and resulting performance are also analyzed.

1.6 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 is a background introduction to the JPEG standard and the DSP-RAM architecture. In the first section, we review the fundamentals of image processing, the structure of the JPEG encoder and the main algorithms used in the JPEG encoder. In the second section, we first introduce the DSP-RAM architecture in detail, and then we review DSP-RAM implementations of key JPEG functions that have been done before.

Chapter 3 presents a parallel JPEG encoder and a parallel DWT algorithm on DSP-RAM. We first discuss opportunities and challenges when using DSP-RAM to implement the JPEG encoder and DWT algorithms. Then we propose hardware and software solutions that solve the challenges. Finally, we describe the process flows and data flows of the parallel implementations in detail.

Chapter 4 presents an FPGA implementation of an improved DSP-RAM on the Xilinx Virtex-II Pro platform. We utilized state-of-the-art FPGA technology, and improved the DSP-RAM architecture at both the component and system-levels. The performance improvement and the bottleneck of the latest DSP-RAM architecture is also analyzed based on the timing and resource report.

Chapter 5 evaluates and analyzes the parallel JPEG encoder implementation presented in chapter 3. We first discuss the platform and methodology that we used for our evaluation. Then we evaluate the overall performance improvements of the parallel JPEG encoder over the sequential JPEG encoder from the IJG Group executed on a Pentium 4 1.6-GHz machine. Based on our evaluation, DSP-RAM's ability to deal with system-level problems is finally analyzed.

Chapter 6 concludes the work and suggests directions for further research.

Chapter 2

BACKGROUND

DSP-RAM is a parallel computing architecture with processor-in-memory and SIMD features. It is designed for high performance and computationally-intensive applications and is intended to solve system-level problems efficiently. To fully evaluate DSP-RAM's ability to solve system-level problems, we chose JPEG as our target because of its realistic complexity and its popularity. Although JPEG 2000 is the state-of-the-art still image compression standard, we chose JPEG because we could more easily find mature and free JPEG codec implementations from industry as a reference for our parallel implementation, and this choice avoids the need to spend time developing it on our own. Nevertheless, because the main difference between JPEG 2000 and JPEG is that JPEG 2000 adopts the DWT (Discrete Wavelet Transform) instead of the DCT (Discrete Cosine Transform) to transform the image from the spatial domain to the frequency domain, we chose the reversible FDWT (Forward Discrete Wavelet Transform) function from the JPEG 2000 standard as another benchmark function so that we could also at least partially investigate DSP-RAM's ability to deal with JPEG 2000.

This chapter reviews the JPEG and JPEG 2000 standards and the DSP-RAM architecture. Particular emphasis is placed on implementations of DCT algorithms. An earlier DSP-RAM implementation of the DCT stage is also introduced and evaluated.

2.1 Reviews of the JPEG and JPEG 2000 Standards

2.1.1 Background

Even in a very small block of space in the real-world environment, there is enormous static and dynamic visual information. The information that people can sense is only a very small part of the total radiated information. The video signal only represents the visible and audible portions of the information and their changes. The images signal only represents the visible part of a video frame at a certain moment.

A small image still contains a great amount of information because of the spatial continuity. However, most of the information is not perceptible to the Human Visual System (HVS). A digital image sampled from the original image signal at a reasonable resolution will be generally sufficient enough for the purpose of our HVS, and the information discarded by the sampling process will not be perceived. For the sake of exchange and ease-of-use, many image file formats have been developed, such as BMP, PPM, PGM, TGA etc. Following a similar argument, a reasonable number of digital images sampled at discrete instants over a period of time should contain sufficient information from the original video signal for the purpose of our HVS.

However, even sampled digital images and digital videos still contain much redundancy for our HVS, and this requires extra storage space and transmission time. For example, a 720 × 480 resolution digital color image with 8-bit pixels needs 1.0368 Mbytes of disk space to store. A 30-second full-motion color video at 30 fps (frames per second) and 720 × 480 resolution needs 933.12 Mbytes disk space to store, and needs more than 1 hour to transmit over a 2-Mbps high-speed Internet channel.

Image compression is a process that yields a more compact digital representation of an image signal by reducing the redundancy of its original representation without significantly losing information or introducing noise that would affect the reconstructed image. A system view of the image compression process is depicted in **Figure 2.1-1**.

An image compression system includes an encoder and a decoder. The core of the encoder is the source encoder, which compress the digital image into a compressed bit stream. The compressed bit stream is translated by the channel encoder into a form that is supported for storage or transmission. The decoders do the reverse process and reconstruct the original image.

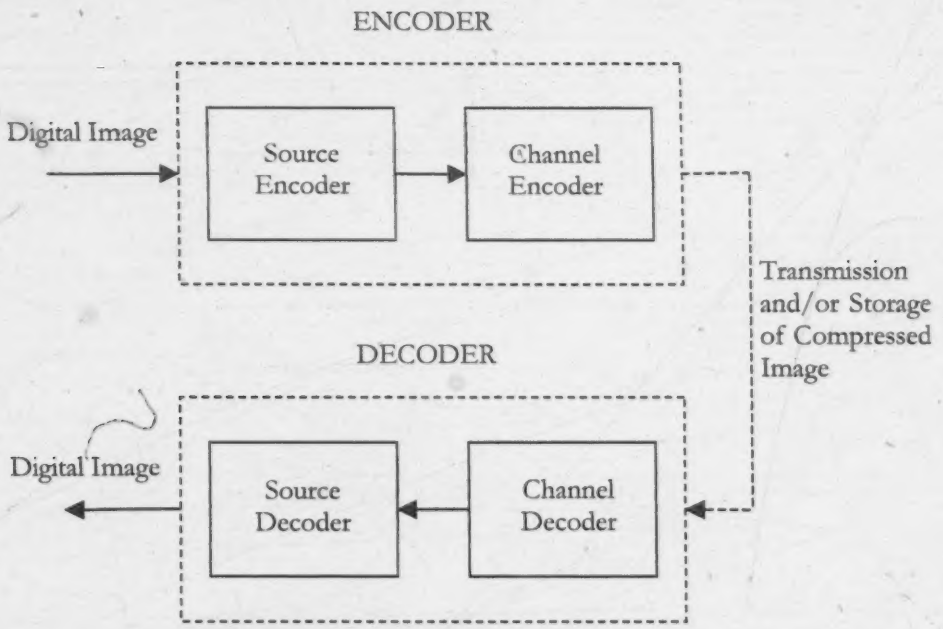

Figure 2.1-1 Generic compression system.

Typical sources of redundancy in digital image signals include:

- *Irrelevant information for the HVS*: information present in digital images that is not perceptible to the HVS.
- *Spatial correlation*: samples within an image or a video frame tend to correlate strongly with their neighboring samples.
- *High-level structure features*: some images have some high-level features that are nearly constant and therefore redundant across regions of space, such as cartoons and text.

Although there are already many methods and algorithms that could be used to reduce the redundancy of a digital image, it is actually fairly complicated to define a set of compression schemes that can be used as an international standard for a wide variety of different applications and users. The choice of a compression method involves a tradeoff along the four dimensions shown in **Figure 2.1-2** [26].

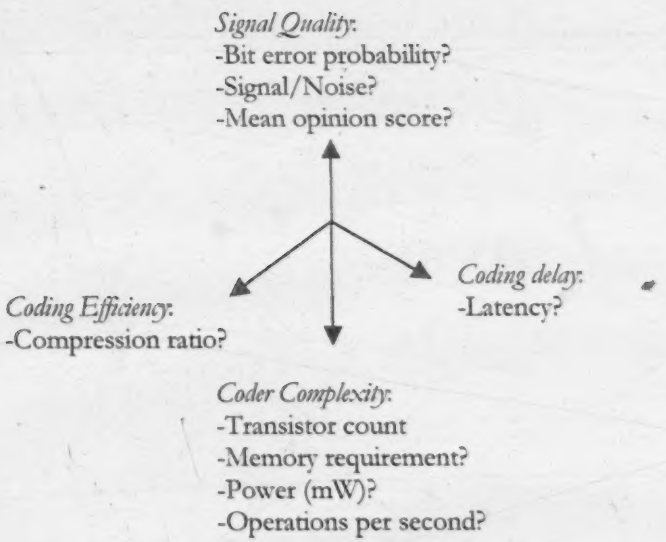

Figure 2.1-2 Tradeoffs in choosing compression methods

In the mid-1980s, The International Telecommunication Union (ITU) and the International Organization for Standardization (ISO) started to work on establishing a joint international standard for the compression of multilevel still images [26]. The experts group became known as the Joint Photographic Experts Group (JPEG).

The goal of JPEG has been to define general methods for image compression that utilize the latest image compression technologies of the time, that allow applications (or a user) to easily trade-off the desired compression rate against the image quality, that work independently of image type, that have modest computational complexity, and that support four modes of operation, namely, sequential DCT-based coding, progressive DCT-based coding, lossless coding and hierarchical coding.

In 1988, JPEG members selected a DCT-based method after evaluating a number of candidate coding schemes. In 1991, the so-called JPEG standard became a draft international standard. In 1992, JPEG became an international standard [27]. After that, JPEG continued working on future enhancements.

JPEG 2000 is a successor to JPEG that is based on wavelet technology rather than the DCT. Its Committee Draft of Part 1 was finished in December 1999. The final draft

international standard was issued in August 2000, and was accepted as a full international standard in December 2000 [27]. As shown in **Figure 2.1-3** [28], JPEG 2000 provides superior lossy compression performance over JPEG at low bit rates below 0.25 bpp (bits per pixel), and significantly reduces the blockiness effect that is evident with JPEG. Another advantage of JPEG 2000 is its error resilience. The JPEG 2000 decoder is better than the JPEG decoder at recovering images after the transmission has suffered packet loss or noise in the bit stream [29].

Figure 2.1-3 JPEG 2000 vs. JPEG

2.1.2 Architecture of the DCT-based JPEG Encoder

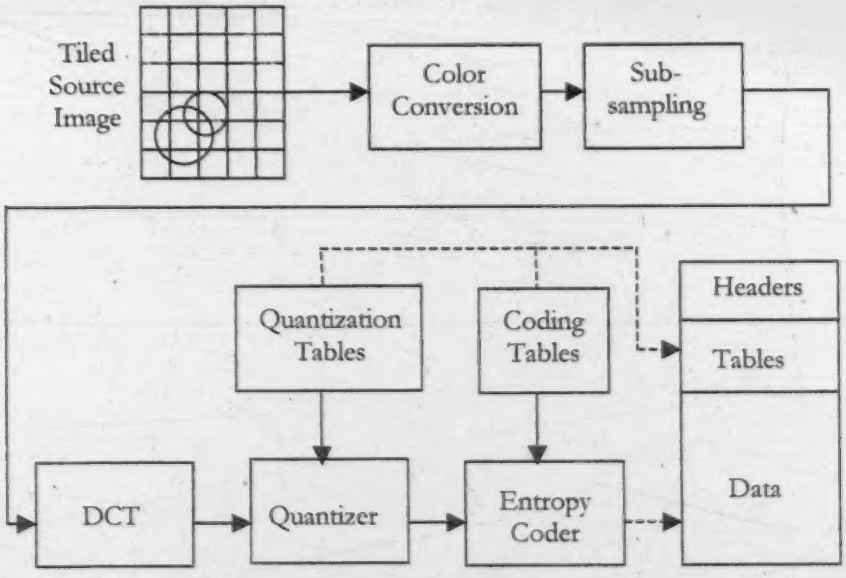

Figure 2.1-4 Block diagram of a JPEG encoder

The architecture of a typical JPEG encoder is shown in **Figure 2.1-4** [26].

Generally, a JPEG encoder is independent of the source file format. The application software is responsible for retrieving image data from some source. The encoder's responsibility is to compress the two-dimensional array of digital image data into a data stream, and then put the data stream into a jpg file. Most jpeg files use the JPEG File Interchange Format (JFIF) [30].

The Color Conversion function in **Figure 2.1-4** transforms the source image data to a color space that has less correlation among its components. The subsampling function removes redundant information from the components that are less perceptible to HVS.

The DCT function then transforms images from the spatial domain to the frequency domain block by block. The quantizer rounds up/down the frequency coefficients to certain discrete integer values. The entropy coder compresses these discrete integer values into a bit stream in two processing stages. The first stage compresses the DC coefficients with a predictive coder, or compresses the zero-valued AC coefficients with a run-length coder. The second stage compresses the remaining non-zero AC coefficients with a Huffman coder or an arithmetic coder. Finally, the compressed bit stream is put into a JFIF file.

The detailed descriptions of these functions are as follows.

Color Conversion

Digital image data can be represented in different color systems, such as RGB, YUV, YIQ, and YCbCr [31]. Some color systems, like RGB, have significant correlation among the color components. When compressing images that are already represented in such color systems, a JPEG encoder may use a color conversion function to transform its representation into a luminance and chrominance representation in either the YUV or the YCbCr color systems which has less correlation among their components. A subsampling function can then be used to delete imperceptible information from components.

For example, for an image in the RGB color space, every pixel is represented by three components which give the weights of Red, Green and Blue color. Every color component actually contains both luminance information and chrominance

information. If we transform the image from the RGB system to YCbCr, the Y component will mainly contain luminance information, while the Cb and Cr components will mainly contain chrominance information. Because the HVS is less sensitive to variations in the chrominance information, we can use fewer samples to represent the Cb and Cr planes without seriously degrading the quality of the reconstructed image. Therefore, we can subsample the Cb and Cr components of the digital image. In this way, we can compress the original image even before we transform the image to the frequency domain.

To transform the representation of an image from RGB to YCbCr, the new component values of all the pixels can be calculated from their old component values through the following equations [32]:

$$Y = (0.29900 \times R) + (0.58700 \times G) + (0.11400 \times B) \qquad \text{Equation 2.1-1}$$

$$C_b = (-0.16874 \times R) - (0.33126 \times G) + (0.50000 \times B) + 128 \qquad \text{Equation 2.1-2}$$

$$C_r = (0.50000 \times R) - (0.41869 \times G) - (0.08131 \times B) + 128 \qquad \text{Equation 2.1-3}$$

These equations are adopted in the JPEG encoder from IJG (Independent JPEG Group, please refer to chapter 3 for more details about the IJG group's JPEG Codec.) and are consistent with the definitions in CCIR[1] 601-1, except that Cb and Cr are normalized to the range 0 – 255 rather than -0.5 – 0.5. The coefficients in these equations are derived from the TIFF standard 6.0 section 21 [33].

From the above equations, we can see that it takes 9 multiplications and 8 additions to process the color conversion of one pixel. Research shows that the RGB to YCbCr color transform may represent up to 39% of the overall compression process for RGB images [26].

[1] Consultative Committee for International Radio

Downsampling

After color conversion, the image pixels are represented by luminance and chrominance components. The luminance component contains most of the spatial information and little perceivable information will be lost if we downsample the chrominance components. If we downsample the chrominance components by a factor of two, the image size will be reduced in half already before we even apply JPEG compression to it.

The downsampling process by a factor of two is illustrated in **Figure 2.1-5**

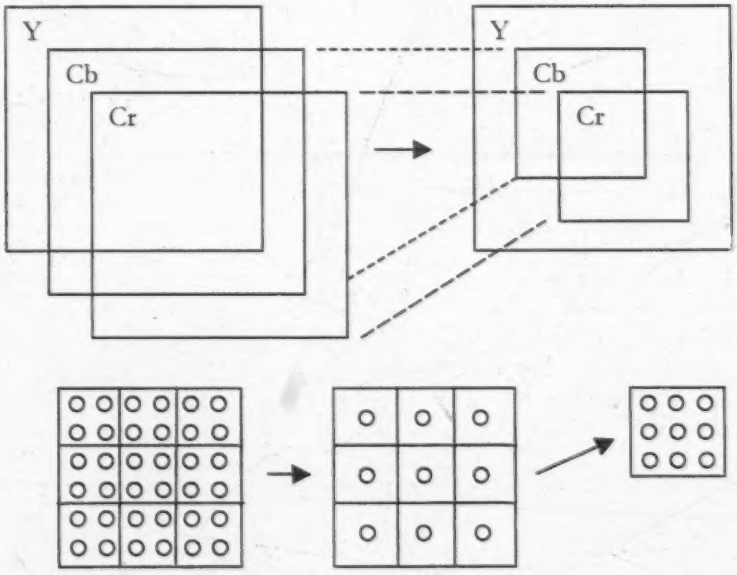

Figure 2.1-5 Illustration of the downsampling process

As illustrated in **Figure 2.1-5**, every four-pixel neighborhood in the Cb and Cr planes is averaged to produce one pixel value in the processed image. The downsampled Cb and Cr planes have only a quarter of their original size, while the Y plane remains intact. The processed image is a half of the original size.

DCT Algorithms

DCT-based image compression is the basis for almost all of the image and video compression standards. The basic computation is to transform an N × N image block from the spatial domain to the frequency domain.

After the preparations of color conversion and downsampling, the image is ready for JPEG compression. First, the image is divided into non-overlapping 8 × 8 blocks. The last column or the bottom row of pixels may need to be duplicated to ensure that the last column or the bottom row of blocks have the same 8 × 8 size. Then the whole image is transformed though a forward DCT algorithm block-by-block.

As with the Fourier Transform, which transforms functions from the time domain to the frequency domain, the DCT transformation projects an 8 × 8 input block from the spatial domain to 64 orthogonal spatial basis functions. The output from the DCT is an 8 × 8 block of basis functions coefficients, which can be used in a weighted sum of basis functions to exactly recover the original input block.

The spatial waveforms for the 64 DCT basis functions are depicted as grey-scale 8 × 8 bitmaps in **Figure 2.1-6**.

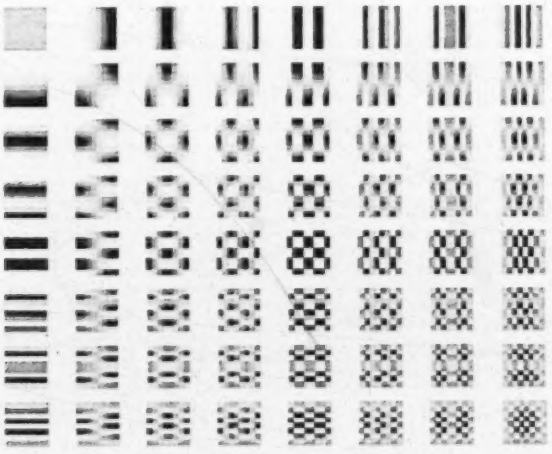

Figure 2.1-6 64 8 × 8 DCT spatial basis functions.

Many DCT algorithms can be used in JPEG compression. To present the factors that affect the choice of DCT algorithms, the following three subsections will introduce the basic 2-D DCT algorithm, the basic 1D DCT algorithm and a fast 1-D DCT algorithm with 12 multiplies and 32 additions.

✧ 2-D DCT

The Basic 2-D FDCT algorithm on 8 × 8 blocks is expressed as

$$y_{kl} = \frac{c(k)\,c(l)}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} x_{ij}\cos\left(\frac{(2i+1)\,k\pi}{16}\right)\cos\left(\frac{(2j+1)\,l\pi}{16}\right) \qquad \text{Equation 2.1-4}$$

where $k, l = 0, 1, \ldots 7$; x_{ij} is the input sample at position (i, j), where i, j = 0, 1, …, 7; and

$$c(k) = \begin{cases} 1/\sqrt{2} & \text{if } k=0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{Equation 2.1-5}$$

The products of cosines correspond to the spatial basis functions.

The equation can be rewritten in simplified form as follows:

$$y_{kl} = \left(a_{kl00} \times x_{00}\right) + \left(a_{kl01} \times x_{01}\right) + \ldots + \left(a_{kl77} \times x_{77}\right) \qquad \text{Equation 2.1-6}$$

Here the coefficients, a_{klij}, where $i, j \in [0,7]$, are assumed to be calculated in advance.

To calculate one frequency coefficient, the basic 2-D DCT algorithm needs 64 multiplications and 64 additions, or 64 MAC (Multiply and Accumulation) operations. To calculate a 2-D DCT on the whole 8 × 8 block, we need 4096 multiplications and 4096 additions, or 4096 MAC operations.

✧ 1-D DCT

The 2-D DCT can also be calculated by first calculating a 1-D DCT on the input block row-by-row, then calculating a 1-D DCT on the resulting intermediate block column-by-column, as depicted in **Figure 2.1-7** [26,32].

The basic 1-D FDCT algorithm is expressed as:

$$z_{il} = \frac{c(l)}{2}\sum_{j=0}^{7} x_{ij}\cos\left(\frac{(2j+1)\,l\pi}{16}\right), \quad i = 0, 1, \ldots, 7 \qquad \text{Equation 2.1-7}$$

where $c(l)$ is defined in Equation 2.1-5.

In matrix notation, the 2-D DCT can be expressed as:

$$Y = T\ X\ T' \qquad \text{Equation 2.1-8}$$

where Y denotes the output matrix in the frequency domain, X denotes the input matrix in the spatial domain, T denotes the transformation matrix, and T' denotes the transpose of T.

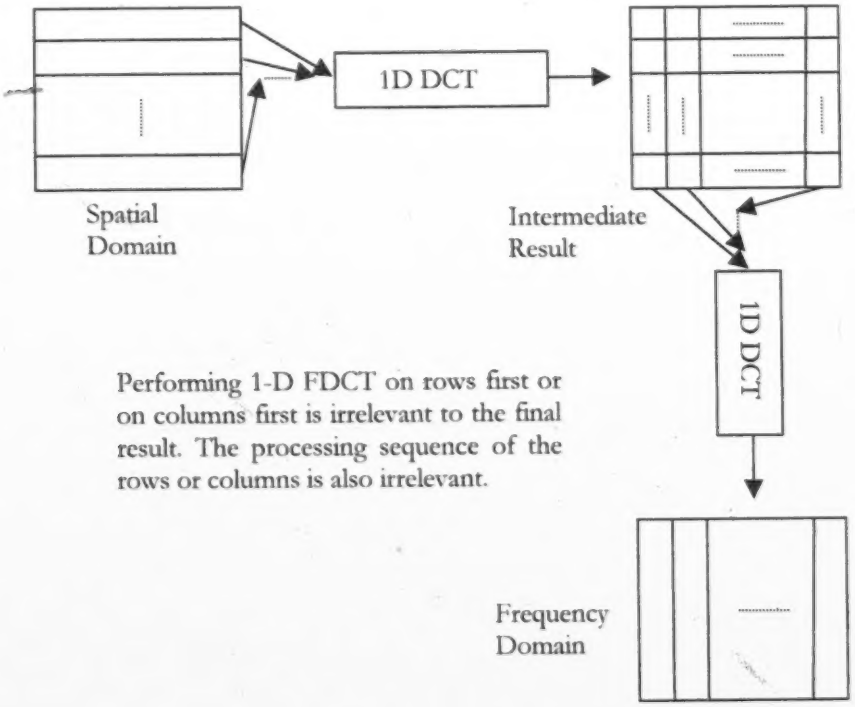
1D DCT
Spatial Domain
Intermediate Result
1D DCT
Performing 1-D FDCT on rows first or on columns first is irrelevant to the final result. The processing sequence of the rows or columns is also irrelevant.
Frequency Domain

Figure 2.1-7 Using the 1-D DCT to compute the 2-D DCT

Using the 1-D DCT to compute the 2-D DCT can then be expressed as

$$Z = T\ X' \qquad \text{Equation 2.1-9}$$

$$Y = T\ Z' = T\ X\ T' \qquad \text{Equation 2.1-10}$$

A 1-D DCT transformation on an 8-pixel row or column needs 64 multiplications and 64 additions, or 64 MAC operations. Sixteen 1-D DCT calculations are required to process an 8×8 block, so the overall operations include 1024 multiplications and 1024 additions, or 1024 MAC operations, which is 4 times less than the basic 2-D DCT algorithm. Hence, most practical software and hardware implementations of DCT-based coders use the row-column extension of the 1-D DCT.

✧ Fast DCT

Most of the fast algorithms for the DCT exploit the properties of the transformation matrix T. The transformation matrix T can be factored so that

$$T = T_1 T_2 ... T_k \qquad \text{Equation 2.1-11}$$

where each of the matrices T_1, T_2, ... T_k is sparse. Sparseness implies that most of the elements of the matrix are zero. So the transform TX can be calculated by first calculating $T_k X$, then $T_{k-1}(T_k X)$, then $T_{k-2}(T_{k-1}(T_k X))$, so on and so forth. The number of operations needed to perform a 2-D DCT can be reduced by choosing an appropriate factorization.

Table 2.1-1 Computational complexity of various fast DCT algorithms

DCT or IDCT Method	Multiplications		Additions	
	1D	2D	1D	2D
1D Chen	16	256	26	416
1D Lee	12	192	29	464
1D Loefller, Ligtenberg	11	176	29	464
2D Kamanger, Rao		128		430
2D Cho, Lee		96		466

Table 2.1-1 [26] shows the computational complexity of several fast DCT algorithms. The first three algorithms are based on the row-column 1-D DCT approaches. The last two algorithms are exclusively for the 2-D DCT. From **Table 2.1-1**, it seems that the fastest 2-D DCT methods need fewer operations than the fastest row-column 1-D DCT methods. However, 1-D fast DCT methods are more popular in practical implementations because the fast 2-D DCT methods generally have following disadvantages:

In software implementations, fast 2-D DCT methods need 128 elements to be stored during the process, which is not always feasible for a register-limited computer system. Data addressing in the fast 2-D DCT methods is generally highly irregular. The additional overhead for address calculation in hardware or software increases the implementation complexity and cost.

Table 2.1-2 Computational complexity of scaled DCT algorithms

DCT or IDCT Method	Multiplications		Additions	
	1D	2D	1D	2D
1D Chen	8	128	26	416
1D Winograd	5	80	29	464
1D Lee	11	176	29	464
2D Kámanger, Rao		92		430
2D Feig, Winograd		54		466

For those compression systems that have a quantizer following the DCT, it is possible to combine the multiplications in the last stages of the DCT with the multiplications in the quantization stage, and further reduce the number of multiplications in the whole process. This method is called Scaled DCT. **Table 2.1-2** shows the computational complexity of scaled DCT algorithms [26].

The main disadvantage of the scaled DCT algorithms is that they are efficient only when the quantization matrix is fixed during the compression, which is the case for still image compression. For video compression, the quantization matrix can be varied at a rate of up to 600 times per second.

IJG's integer DCT implementation is based on C. Loeffler and A. Ligtenberg 's paper "Practical Fast 1-D DCT Algorithms with 11 Multiplications" [34]. The primary algorithm introduced in the paper uses 11 multiplications and 29 additions. For convenience, the IJG uses another algorithm that requires 12 multiplication and 32 additions.

Quantization

The coefficients matrix obtained from the DCT transformation still contains unimportant information in most of the high frequency components. Depending on

the required quality of the reconstructed image, we can discard a certain portion of this information to gain a higher compression rate.

After quantization, many of the small high-frequency components are replaced by zeros, which can then be compressed greatly by using run-length coding in a zigzag sequence explained later. **Figure 2.1-8** shows a typical coefficients matrix after quantization.

89	11	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 2.1-8 A typical coefficients matrix after quantization

Entropy Encoding

The last processing stage in JPEG is entropy encoding. This stage involves differential coding for the DC coefficients, followed by Huffman coding for the DC differential values, run-length coding for all the zero-valued AC coefficients, and Huffman coding for all the non-zero AC coefficients.

✧ Differential coding

Typically the DC coefficients are much greater than the AC coefficients, thus they need more bits to represent. Due to the relatively high correlation of the DC values among adjacent blocks, JPEG represents them compactly using differential coding.

✧ Run-length coding

After quantization, most of the elements of the coefficients matrix become zero. Furthermore, for most images, the non-zero coefficients tend to be clustered at the

left-top corner of the coefficient blocks. Thus, JPEG uses run-length coding in a zigzag order to compress the zero-valued AC coefficients. **Figure 2.1-9** depicts the zigzag ordering in an 8×8 matrix.

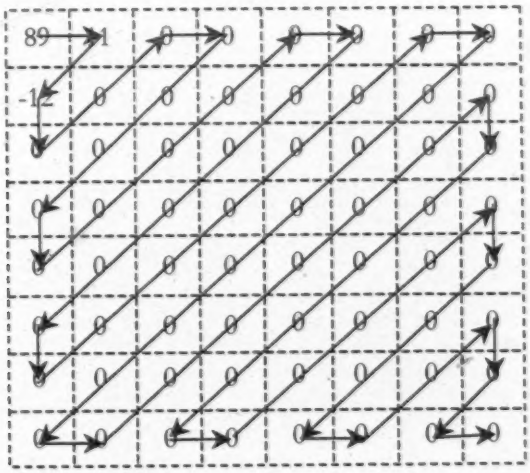

Figure 2.1-9 Zigzag ordering in an 8×8 matrix

✧ Huffman coding

For the differential DC values and the AC values, JPEG adopts Huffman coding or arithmetic coding to perform lossless compression. Generally, arithmetic coding can achieve a higher compression rate and no look-up table needs to be sent along with the compressed data. However, compared to Huffman coding, the improvement in the compression rate of arithmetic coding is more than offset by the increase of its computational complexity; hence, Huffman coding is adopted for the most applications.

The idea behind Huffman coding is, instead of representing all of the values in a fixed bit-length such as 8 bits, to use fewer bits to represent the values that appear more frequently, and to use more bits to represent the values that appear less frequently. Thus on average the number of bits used to represent a coefficient in a set of coefficients is close to the entropy of the coefficients set, which is usually much less than 8 bits. In this way, the quantized DC and non-zero AC coefficients are further compressed without losing any information [26].

The Huffman encoder defined in the JPEG standard is not the straightforward implementation of Huffman coding methods. Instead it employs one or more standard Huffman look-up tables to simplify the process. For any DC differential value, the Huffman encoder first represents it in a pair of (precision, amplitude) values, and then replaces the precision value with a codeword in the corresponding Huffman look-up table. The precision is the number of bits needed to represent the amplitude. The amplitude is computed according to the DC differential value: if it's positive, its binary representation with given precision is used directly; if it's negative, we first get its absolute value, and then compute the one's complement of the absolute value as the amplitude. For example, for a DC differential of 5, instead of using its 8-bit binary representation 00000101, we use a (precision, amplitude) pair to represent it as (3, 101). If the Huffman codeword for precision =3 is 10, then the DC differential 5 is coded as 10101. Similarly, -5 would be coded as 10010.

For a non-zero AC coefficient, it is represented as a pair (run/precision, amplitude). The run value gives the number of zero AC coefficients preceding this one; the precision and the amplitude are computed as in the case of DC differentials. The value of run/precision is Huffman coded by using lookup tables. For example, a non-zero AC coefficient -12 preceded by 9 zeros would be represented as (9/4, 0011). If the Huffman codeword for (9/4) is 11110011, then the AC coefficient -12 and the 9 preceding zeros would be coded as 111100110011.

JPEG does not define default Huffman tables, but most applications use the Huffman tables listed in the standard.

Entropy coding involves many shift operations and address calculations that might be difficult to implement efficiently on the DSP-RAM architecture. As we will introduce later, up to 14% of the overall processing time of IJG JPEG encoder is spent on entropy coding. For current DSP-RAM architecture, which lacks support for shift operations and address calculations, it will take even more time to process.

2.1.3 The Discrete Wavelet Transform in JPEG 2000

A major difference between JPEG 2000 and JPEG is the way that they transform images between the frequency domain and the spatial domain. The Discrete Wavelet

Transform (DWT), which is adopted by JPEG 2000, provides a more efficient way to represent both the frequency and spatial information of images.

Figure 2.1-10 shows an example of the operation of a DWT filter bank. The π in the figure stands for the maximum frequency in an analog image signal that we can reconstruct from the sampled pixels according to Shannon's Sampling Theorem. For example, suppose we have eight pixels and the sampling rate is 8 per unit length, and then the π is 4 per unit length.

Consider the eight pixels as a sampled signal in the spatial domain. In the first level, we pass the signal through a high-pass filter g[n] and a low-pass filter h[n], and get two output signals, each with 8 samples. For the simplicity of demonstration, we assume that the high-pass filter and low-pass filter chosen in **Figure 2.1-10** always cut the frequency band of the input signal exactly in half. In real-world implementations, the cut-off frequency can be adjusted by choosing different filters.

Therefore, the signal outputted from the high-pass filter contains only those high frequency components of the original signal in the frequency band from $\pi/2$ to π. Since it is actually a signal in the spatial domain, it also contains the spatial information of the original input signal. For example, if the first four pixels of the input signal form a pulse signal, then the corresponding high-frequency components will appear in the first four samples of the output signal. Similarly, the signal outputted from the low-pass filter also contains those low frequency components of the original signal in the frequency band from 0 to $\pi/2$ and their distribution information. Since we are only interested in the frequency bands from 0 to $\pi/2$ or from $\pi/2$ to π for the output signals, we can subsample the results by a factor of 2 to reduce the sampling rate in half without loss any information. The 4 resulting samples from the high-pass filter are called level 1 DWT coefficients, and the 4 outputs from the low-pass filter are passed to the next level for processing.

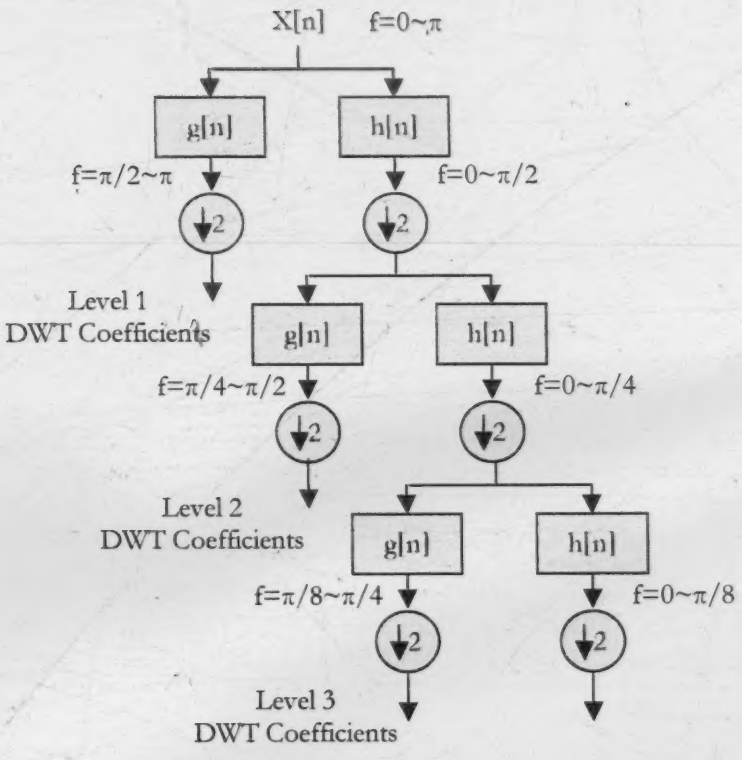

Figure 2.1-10 Sub-band coding decomposition of DWT

As shown in **Figure 2.1-10**, we continue to pass the outputs of low-pass filter to the next level and then divide the frequency band into smaller frequency spans until we have only two samples. In our case, it takes 3 levels of processing for the full decomposition of 8 pixels. The results contain one coefficient each representing the weight of frequency components in the intervals $[0, \pi/8)$ and $[\pi/8, \pi/4)$, two coefficients representing the weights of frequency components in $[\pi/4, \pi/2)$, and 4 coefficients representing the weights of frequency components in $[\pi/2, \pi]$.

Although DCT-based coding schemes offer reasonable rate-distortion performance in mid or high bit-rates, Wavelet-based coding schemes are more robust under transmission or decoding errors and generally offer better rate-distortion performance at low bit-rates (e.g., below 0.25 bpp) [35]. And since it's not necessary to block input images, the blocking artifacts at high compression rates are avoided. And its inherent

multi-resolution and progressive coding nature also facilitates the progression transmission of images.

2.2 Review of DSP-RAM

DSP-RAM is a directly scalable programmable parallel architecture that combines the ideas of DSP, SIMD and processor-in-memory. It could be a very efficient platform for computationally-demanding applications that require many multiplications and additions but have a limited power budget.

In this section, we will first give a brief introduction to the DSP-RAM architecture, and then we will introduce past research that has been done on still image compression.

2.2.1 The DSP-RAM Architecture

Top-level Architecture

DSP-RAM's top-level architecture incorporates features of two other architectures: SIMD and processor-in-memory. DSP-RAM uses the SIMD architecture to organize and operate its PEs, and uses the processor-in-memory architecture to achieve the high memory bandwidth required by many high performance applications. Both features help DSP-RAM to achieve high performance with low power consumption.

The SIMD architecture of DSP-RAM enables it to distribute computational tasks to its PEs for parallel execution. The architecture could provide sufficient computational power at a relatively low working frequency compared to a conventional superscalar microprocessor. Thus power consumption can potentially be much lower.

The integration of processor and memory on a single chip avoids the power consumption needed by conventional microprocessor systems to drive signals off-chip. It also avoids the limitation on the width of the memory data bus which mainly results from the pin number limitation of available chip package.

Figure 2.2-1 shows the DSP-RAM's top-level architecture. The host controller, which could be a conventional microprocessor on the same chip or on another chip, operates all the processing elements through the global instruction bus in a SIMD manner. Each PE has dedicated access to two local memory banks through two separate local memory buses. The host controller can also access any memory location in the conventional way

through global data and address buses for the sake of data preparation, data organization and data sharing among processing elements. Data can also be shared among processing elements through the one-dimensional shift register network which connects all the processing elements into an array.

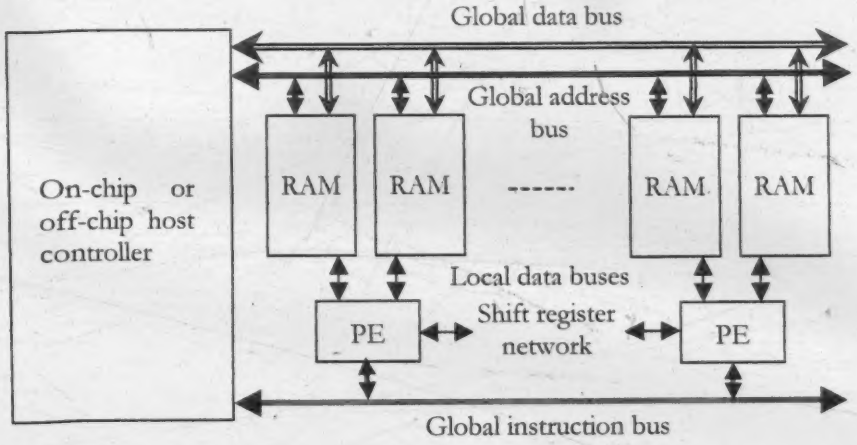

Figure 2.2-1 DSP-RAM's top-level architecture

DSP-RAM's top-level architecture is highly scalable. The size of the local memory blocks and the number of PE can be adjusted at design time using VHDL generics according to the need of applications and the transistor budget. To avoid long wires, all the global buses and clocks should be routed in a tree-structured manner with short distances from the route nodes to the leave nodes at the control inputs.

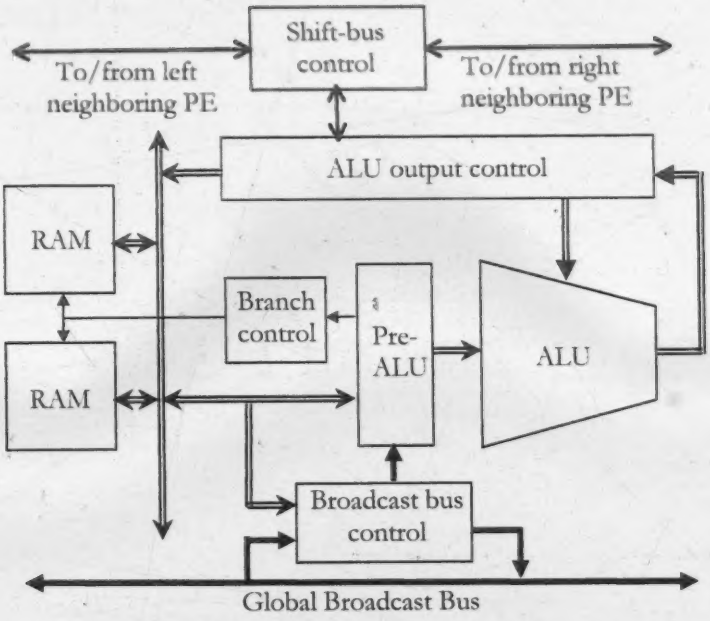

Figure 2.2-2 Diagram of DSP-RAM's PE architecture

DSP-RAM's PE design embodies a tradeoff between processing power and size. Generally, more processing power per PE means more transistors will be needed for every single PE and thus fewer PEs can be integrated into one single chip. Fewer PEs means less parallelism and may imply less processing power at chip level depending on how effectively the processing power of each PE can be applied.

In large vector super-computers, like the Cray-I [36,37], the ALU of PEs could have specialized processing components for logical, integer and floating-point computations. But in the case of DSP-RAM, the architecture of the PE has to be simplified to permit greater SIMD parallelism on a single chip. **Figure 2.2-2** depicts DSP-RAM's PE architecture diagram and the data path.

The PE of the existing DSP-RAM deploys a pipelined integer MAC (Multiplication and Accumulation Control) followed by a shift register as its ALU. The Pre-ALU block in **Figure 2.2-2** controls the inputs of ALU: operands may be routed in from two memory banks or broadcast bus. The ALU output control block controls the output of

the ALU: results can be written back to local memory or to the left-right shifter register network.

Figure 2.2-3 illustrates the architectural details of the PE in DSP-RAM. There are many features that could be conditionally included in the PE architecture using VHDL generics to meet the particular requirements of applications. Optional features include the branch control realized by a stack and a comparator; the global compare realized by an arbiter along with the broadcast bus, the shift register (the Sh Reg in the **Figure 2.2-3**) that scales data before it is transmitted to the left-right shift register, etc.

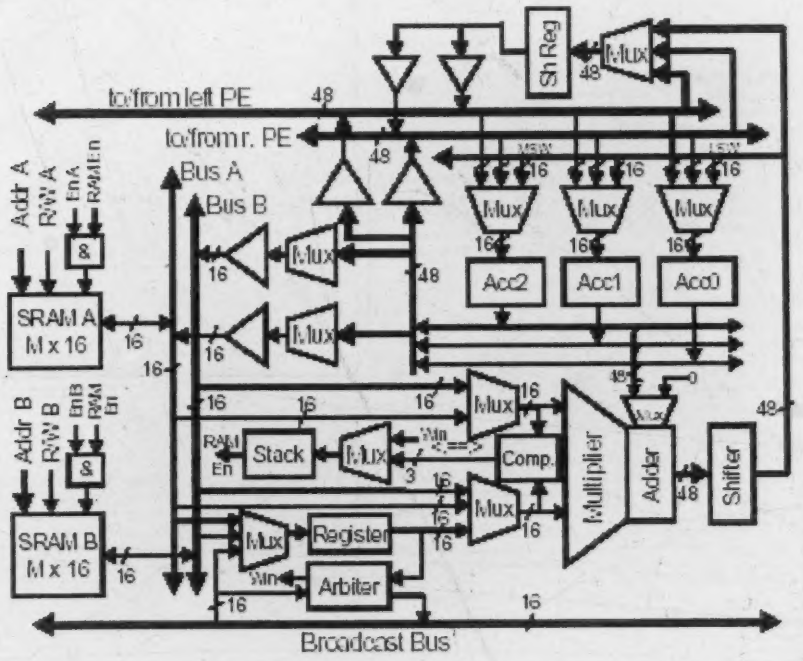

Figure 2.2-3 Detailed PE architecture

ALU Architecture

As depicted in **Figure 2.2-3** [8], DSP-RAM's ALU includes a 16-bit multiplier, a 48-bit accumulator and a 48-bit shifter. The pipelined multiplier multiplies two 16-bit inputs. The 32-bit product is accumulated with a 48-bit input that can be fed back from the output of the ALU or from a neighboring PE. The sum of the adder is passed through a 48-bit shifter. The 48-bit internal data path of the ALU enables the PE to perform up to 65,536 MAC operations without overflow. The shifter can also be used to increase the

number of MAC operations before overflow and adds flexibility to the ALU to perform scaling operations on intermediate or final results from the MAC.

2.2.2 Past Research

Several standard algorithms in DSP, wireless communications and multimedia areas that have been implemented and evaluated on the DSP-RAM architecture [6,7,8,9]. In this section, we will introduce Billie Kwan's parallel 2-D DCT implementation on DSP-RAM. We will evaluate its efficiency and discuss its advantages and disadvantages. Based on the discussion, we will propose a new way to implement the DCT in the next chapter.

Billie Kwan's 2-D DCT Implementation

Billie's 2-D DCT implementation uses the DSP-RAM as a big pipeline, feeding pixels into the pipeline and outputting coefficients with a throughput of one coefficient per clock cycle.

As we described in section 2.1.2, to calculate one frequency coefficient with the 2-D DCT algorithm, we need to perform MAC operations on two 8×8 matrixes: the pixel matrix and the spatial domain basis functions matrix.

Billie's implementation uses 64 PEs as 64 stages of a big pipeline that is connected by the left-right shift register network. The 64 spatial domain basis functions matrixes that are needed for the calculation of 64 frequency coefficients are stored in the local memory of the 64 PEs, and the input pixel stream is fed into the pipeline over the broadcast bus one pixel per clock cycle. Thus the 64 PEs can perform MAC operations on the available data calculating all of the 64 frequency coefficients at the same time. The latency needed for calculating one frequency coefficient on one PE is 64 clock cycles plus the latency of one PE, which is 5 clock cycles in the current version of DSP-RAM. Since the frequency coefficients are calculated in parallel, the latency needed for DSP-RAM to perform the 2-D DCT transformation on a whole 8×8 block will be still 64 clock cycles plus the latency of one single PE. Latency of one PE can be hidden if multiple blocks are processed one after another. Coefficients can be shifted out as new coefficients are computed in the ALUs.

Using Billie Kwan's implementation, an image can be transformed continuously by passing all of the pixels of the image through the pipeline in a certain order and the throughput will be one coefficient per clock cycle. **Figure 2.2-4** shows the data flows in Billie's parallel 2-D DCT implementation.

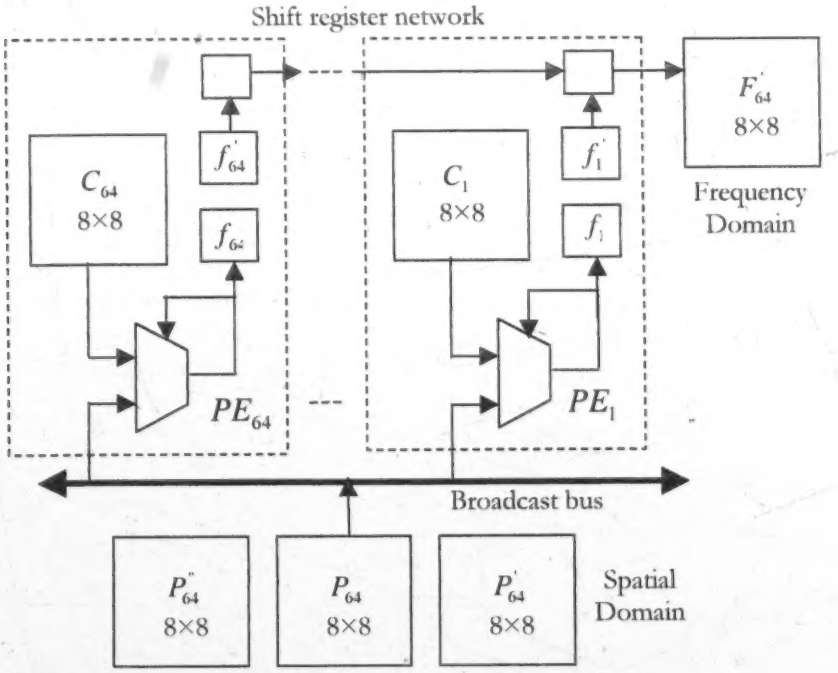

Figure 2.2-4 Billie Kwan's parallel 2-D DCT algorithm on DSP-RAM

The major problems of this scheme are scalability and flexibility. First, even when we have more than 64 PEs on a DSP-RAM, this scheme can only utilize 64 PEs for 8×8 block DCT transformation. Second, if number of PEs is less than 64 and it is not a power of 2, then the hardware utilization will be reduced. For example, if we have only 40 PEs on a chip, we can process the 2-D DCT in two rounds with Billie's algorithm. In the first round we could calculate 40 coefficients; in the second round we could calculate the remaining 24 coefficients. Then in the second round 40% PEs would be idle. The overall utilization of the hardware would then be roughly 80%. Furthermore, Billie's algorithm needs to put the 64 spatial domain basis functions matrixes in memory initially. If the DSP-RAM is not used as a dedicated DCT transformer, the cost of reloading the basis functions matrixes would further degrade the efficiency of Billie's

algorithm. A more efficient utilization for Billie's algorithm in the case of JPEG processing is to process the DCT transform over the whole image so that the initialization step only needs to be done once. But in this way, the quantization and entropy encoding would have to wait until after the DCT transformation of the whole image, instead of only a block of the image, is completed.

Although Billie's DCT algorithm achieves a throughput of one coefficient per clock cycle, it is not the most efficient algorithm for JEPG from a system point of view. In Chapter 3, we will introduce a more flexible, more scalable and more efficient DCT algorithm for the DSP-RAM architecture that can achieve a throughput of several coefficients per clock cycles.

Chapter 3

IMPROVED JPEG AND DWT IMPLEMENTATIONS

This chapter presents our parallel JPEG and DWT implementations on DSP-RAM. In section 3.1, we analyze the general challenges for DSP-RAM when dealing with system-level problems, and discuss potential solutions to solve the challenges. In section 3.2, we analyze alternative ways of implementing JPEG on DSP-RAM from the system-level to the function-level. These ways are illustrated by the detailed process flow and data flow of our parallel JPEG implementation. In section 3.3, we consider the challenge of implementing DWT on DSP-RAM and describe detailed parallel algorithm.

3.1 System-level Challenges for the DSP-RAM Architecture

To investigate DSP-RAM's ability to deal with system-level applications, we first need to study the range of operations that can be solved efficiently on DSP-RAM, which leads us to understand the feasibility of using the DSP-RAM architecture to solve system-level problems. Then we need to analyze the factors that affect the overall performance of a DSP-RAM implementation of a target application, which leads us to understand the possible efficiency that we might gain by using DSP-RAM. We also need to analyze the factors that affect the programmability of DSP-RAM, which leads us to understand the difficulties faced by a developer when using DSP-RAM to solve problems and to investigate possible ways to improve the usability of DSP-RAM.

In the following three subsections, we address these issues accordingly.

3.1.1 DSP-RAM's Functional Versatility

DSP-RAM is not designed to be a general-purpose architecture. Rather, it is intended to be a co-processor that boosts the performance of specific applications. In order to integrate as many as PEs on one chip, DSP-RAM's functionality needs to be adapted to suit the requirements of the target application.

The current version of DSP-RAM is intended for arithmetically-intensive applications. It adopts a simple pipelined MAC as its ALU; hence it is most appropriate for

multiplication-intensive operations on data. Because the host controller of DSP-RAM can also access the memory of DSP-RAM, DSP-RAM can work together with a host controller to process the data in its memory.

Table 3.1-1 [22] lists operations supported by a typical DSP-RAM system. The host controller can be chosen according to the needs of the target applications. Because almost all modern microprocessors support the first four categories of operations, we can safely assume that the host controller that we choose will also fully support them.

Table 3.1-1 List of operations supported by a typical DSP-RAM system

Operations	Host Controller	DSP-RAM	Examples
Arithmetic	Supported	Partially [1]	Integer arithmetic operations: add, multiply, etc.
Logical	Supported	None	Logical operations: and, or, etc.
Data transfer	Supported [2]	Basic [3]	Memory access, data broadcast, shift register network.
Control	Supported	None	Branch, jump, procedure call and return, etc.
System	Optional	None	Operation system calls, etc.
Floating-point	Optional	None	Floating-point operations: add, multiply, etc.
Decimal	Optional	None	Decimal add, multiply, etc.
String	Optional	None	String move, compare, search, etc.
Graphics	Optional	None	Pixels operations, compression/decompression, etc.

1. The pipelined MAC ALU provides hardware support for 16-bit addition, subtraction, multiplication and MAC. There is no direct support for division.
2. The latest version of DSP-RAM adopts a dual-port SRAM as its local memory. Theoretically, the memory could be accessed by the host controller in all addressing modes, just as if it were in the host controller's main memory.
3. The current version of DSP-RAM only supports the direct and absolute addressing modes.

Those operations that can be performed in a SIMD manner and supported by the PEs of DSP-RAM can be performed in parallel on DSP-RAM; the other operations can be performed sequentially on the host controller.

3.1.2 Factors Affecting Overall Performance

As we discussed in section 3.1.1, only some parts of most applications will be sped up using DSP-RAM. The overall speed-up depends on what fraction of the whole application can be efficiently parallelized for DSP-RAM.

Amdahl's Law:

According to Amdahl's Law, the overall application-level speedup depends on the fraction of the application that is sped up and the speed-up for that portion. The law can be expressed as Equation 3.1-1 [22].

$$Speedup_{overall} = \frac{Execution\_time_{old}}{Execution\_time_{new}}$$

$$= \frac{Execution\_time_{old}}{Execution\_time_{old} \times \left(\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)}$$

$$= \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Equation 3.1-1

In the case of the DSP-RAM architecture, when we need to speed up a portion of a calculation, we may introduce new workload to the system because of data initialization, context switches, etc. Hence Amdahl's Law should be modified as Equation 3.1-2:

$$Speedup_{overall} = \frac{Execution\_time_{old}}{Execution\_time_{new}}$$

$$= \frac{Execution\_time_{old}}{Execution\_time_{old} \times \left(\left(1 - Fraction_{enhanced} + Fraction_{overhead}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)}$$

$$= \frac{1}{\left(1 - Fraction_{enhanced} + Fraction_{overhead}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Equation 3.1-2

According to the Equation 3.1-2, we should keep the overhead as small as possible to achieve the maximum performance improvements.

To increase the accelerateable fraction in the DSP-RAM implementation of target applications, we should increase the functional versatility of the DSP-RAM's PEs so that DSP-RAM can undertake more processing tasks in the overall application.

The new workload introduced by utilizing DSP-RAM to speed up a portion of the application is algorithm-dependent. Much extra work appears in the data preparation, or data rearrangement required by the next functional blocks. We therefore need to carefully design the process flow and data flow in the target application to avoid unnecessary data transfers and processing.

Speed up for the Portion of the Application Implemented on DSP-RAM.

For the accelerateable portion of an application, the resulting performance depends more on architectural issues. The execution time or CPU time for a program can be expressed as [22]:

$$CPU\_time = IC \times CPI \times Clock\_cycle\_time \qquad \text{Equation 3.1-3}$$

where IC stands for instruction count, CPI stands for clock cycles per instruction.

The main advantage of the SIMD architecture compared to the SISD architecture is that the SIMD architecture can process the same operations on multiple data streams at the same time, hence reducing the IC needed to process the data. The best possible CPI of a pipelined data path is 1; however, in fact the CPI of a pipelined data path is always greater than 1 due to structure hazards, data hazards and/or control hazards (defined later). Clock cycle time is not only related to architectural issues, but is also related to implementation issues. In general, the process technology, the depth of the pipeline and the balance among difference pipeline stages will all affect the clock cycle time of a microprocessor architecture.

So how can the IC be reduced on the DSP-RAM architecture? The answer depends first on the number of PEs that the DSP-RAM has, which is related to how much data parallelism DSP-RAM can exploit. As DSP-RAM is a highly scalable architecture, the number of PEs in a DSP-RAM chip is mainly limited by the available transistor budget. The answer also depends on the scalability of the implementation of the target application, which is related to how many PEs can be utilized. As we discussed in

Section 2.2.2, a DSP-RAM implementation with less scalability will limit its ability to fully utilize DSP-RAM's computational power.

Like most current microprocessor architectures, DSP-RAM uses pipelining to reduce its CPI. However, unlike conventional architectures, the current version of DSP-RAM do not also pipeline the associated instruction stream. Because the data flowing through the data path does not contain any instruction information, the pipeline can not be operated automatically as a black box. The pipeline has to be controlled by a separate instruction stream which takes full control of all stages. This results in a relatively poor programmability. We will further discuss this later in Section 3.1.3.

CPI is actually a statistical quantity obtained by dividing the number of clock cycles needed to execute a program by the overall instruction count of the program. We will now consider some of the factors that affect the CPI of DSP-RAM implementations.

For a non-pipelined data path, an instruction can be started only after the previous instruction has finished. The CPI of an instruction gives the clock cycles required by the instruction to go through the whole data path. When the instruction goes through one stage of the data path, the remaining stages of the data path remain idle. Therefore, the utilization of hardware is very low. Pipeline technology utilizes the parallelism represented by the different stages of the data path. An instruction can start to use a data path stage once the previous instruction finishes using it. Because the different stages of the data path can be utilized at the same time, the overall hardware utilization is increased. If we neglect the initial time for instructions to fill the pipeline, the CPI for an ideal pipeline is 1, i.e. for every clock cycle, one instruction is being finished by the pipeline. It is easy to see that, as the number of the pipeline stages is increased and the logic depth of each stage is reduced, the throughput and the latency of the pipeline are increased.

Computer instructions are generally designed under the assumption that they will be executed in order and one instruction starts only after the previous one finishes. Simply pipelining the original data path breaks the assumption by starting an instruction before the completion of the previous one, and this opens up that possibility of problems that are called hazards [22]. For example, when an instruction needs to use the result of the

previous instruction as an input augment, the previous instruction may be still in the pipeline and the final result is not yet available. At this time, if this instruction is an ALU operation, then a *data hazard* occurs; if this instruction is a branch or jump instruction, which is used to control the process flow of a program, then a *control hazard* occurs. As another example, sometimes the data path is not completely pipelined in an effort to reduce hardware resources or control complexity. In this case, it's possible that there are two instructions in different stages of the pipeline that need to use the same resource in the data path, such as the same register file, or the same memory locations. If these resources can not be shared, then a *structure hazard* occurs.

Although much research has been done to eliminate the effects of hazards, in some situations we still have to add bubbles into a pipeline to resolve the problem. A *pipeline bubble* means there is one clock cycle in which the data path will produce no result. This leads to a lower efficiency and a higher CPI. For example, if 30% of the instructions introduce one bubble due to hazards, from Equation 3.1-3, the overall CPU time will be 1.3 times longer than the CPU time obtained by executing on an ideal pipeline that somehow requires no bubbles.

DSP-RAM can have both data hazards and structure hazards in its pipeline. To reduce the average CPI of an application on DSP-RAM, we need to find out how these hazards happen so that we can try to avoid them. As for control hazards, because the DSP-RAM architecture put its program flow control on its host controller and the control is not dependent on the results calculated on PEs, this problem is beyond the scope of this thesis.

Figure 3.1-1 shows a simplified view of DSP-RAM's pipeline. DSP-RAM's data path is divided into six pipeline stages by five registers: BB, M0, M1, M2, and ACC. We name the first five stages with the name of the pipeline register following them. The last stage will be called WB, which stands for write-back.

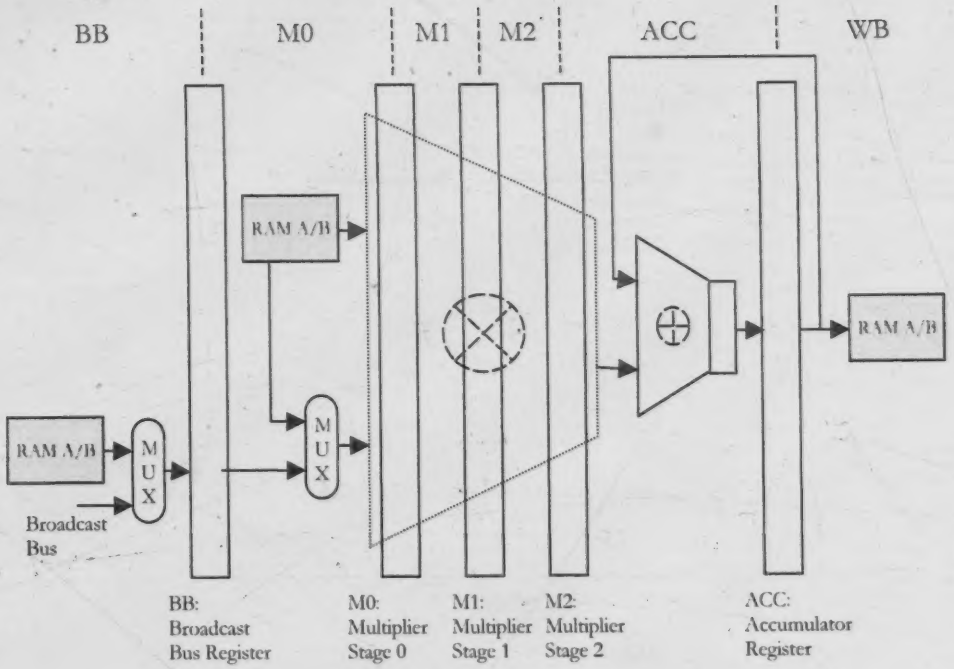

Figure 3.1-1 Simplified DSP-RAM pipeline

As we discussed before, the data flowing through the data path does not contain any control information: the host controller needs to issue instructions cycle-by-cycle over the instruction bus to control the operation of the pipeline. Each instruction issued by the host controller has to contain all of the control information for all stages of the data path.

From **Figure 3.1-1** we can see that there is no hardware constraint preventing the BB, M0, WB stages of data path from accessing memory banks at the same time, but the hardware actually has only two memory banks that can be accessed independently. In other words, any instruction that enables any two stages of data path to access the same memory bank at the same time will cause a structure hazard.

It's possible to eliminate this kind of structure hazard from the DSP-RAM architecture through hardware approaches, such as adding one more bank of memory, or adopting memory interface technologies that allows read and write accesses in the same clock cycle. Another solution would be to transform the current memory-register architecture

into a register-register architecture so that only one stage of the data path is responsible for all memory accesses. However, these solutions imply significant architecture changes that may cause other problems.

An alternative approach is to carefully design an instruction set for DSP-RAM that avoids providing instructions that cause structure hazards. In this way, although structure hazards still exist in the DSP-RAM architecture, they will never occur because of the constraints of the instruction set. If we take a close look at the problem, we find that the following rules need to be obeyed in the design of a structure-hazard-free instruction set for DSP-RAM.

1. When an instruction contains a write-access to one memory bank in the WB stage, then that memory bank cannot also be accessed by the BB stage or the M0 stage. This is because, when one memory bank is used for the output of the instruction, the two operands of the instruction have to be inputted from the other memory bank and the broadcast bus. Since the current version of DSP-RAM can read the broadcast bus only in the BB stage, we should use the BB stage to retrieve one operand from the broadcast bus and use the M0 stage to retrieve the other operand.

2. When an instruction does not contain a memory access in the WB stage, then the only limitation on the instruction is that the memory banks being accessed in BB and M0 stages can not be the same.

Now we will take a look at the data hazards in DSP-RAM architecture. Unlike structure hazards caused by competition for hardware resources, data hazards are caused by accessing memory in the wrong order, therefore generating incorrect results.

For example, suppose that we need to perform the following scalar computations on DSP-RAM:

A+B=C; C+D=E

Table 3.1-2 shows the contents of the DSP-RAM pipeline when a data hazard is ignored.

Table 3.1-2 Calculation on DSP-RAM without considering data hazards

Clock Cycles	Pipeline stages						Comments
	BB	M0	M1	M2	ACC	WB	
CC0	1						
CC1	1	A×1					
CC2	1	B×1	A×1				
CC3	1	**C**×1	B×1	A×1			Old C enters the pipeline.
CC4		D×1	**C**×1	B×1	A		
CC5			D×1	**C**×1	A+B		
CC6				D×1	**C**	**C**	New C loaded back to memory.
CC7					**C**+D		
CC8						**E**	Wrong result (uses old C).

Table 3.1-3 shows how the data hazard can be avoided in the same pipeline by introducing bubble cycles that temporarily occupy time slots in the pipeline.

Table 3.1-3 Calculation on DSP-RAM that avoids data hazards

Clock Cycles	Pipeline stages						Comments
	BB	M0	M1	M2	ACC	WB	
CC0	1						Start A×1
CC1	1	A×1					Start B×1
CC2	Bubble1	B×1	A×1				Start Bubble1
CC3	Bubble2		B×1	A×1			Start Bubble2
CC4	Bubble3			B×1	A		Start Bubble3
CC5	Bubble4				A+B		Start Bubble4
CC6	1					**C**	C available in this clock cycle.
CC7	1	**C**×1					Bubble1 exits pipeline.
CC8		D×1	**C**×1				Bubble2 exits pipeline.
CC9			D×1	**C**×1			Bubble3 exits pipeline.
CC10				D×1	**C**		Bubble4 exits pipeline.
CC11					**C**+D		Addition
CC12						E	Correct result.

Because the result of the second addition (E=C+D) depends on the result of the first addition (C=A+B), the second addition must be performed after the first addition has finished, as **Table 3.1-3** shows. If we start to issue the instruction for the next addition

before the first addition finishes, as **Table 3.1-2** shows, we will get a wrong result due to the data hazard.

To avoid the data hazard in the example above, four bubbles or stalls are introduced into the pipeline to avoid the data hazard. If we neglect the clock cycles needed to fill up the pipeline (CC0-CC4), the overall number of clock cycles for two additions is 8 (CC5-CC12). Therefore the average number of clock cycles per addition is 4. To improve the performance, the bubbles can be replaced by other independent instructions by reordering instructions at compile time. For scalar computing, because the scalar variables are generally accessed relatively randomly, it is usually much harder to avoid data dependencies between instructions. For vector computing, however, it is easier to predict and avoid data dependencies because the data access pattern is much more regular. It is then possible to avoid many data hazards between instructions by carefully designing the data processing flows.

For example, suppose that we need to perform following vector computations on a PE of DSP-RAM:

$$A[i]+B[i]=C[i]; \qquad C[i]+D[i]=E[i]; \qquad \text{where } i\in[0,n]$$

We could perform the two vector additions in following sequence with inserted bubbles:

$$A[0]+B[0]=C[0]; \qquad C[0]+D[0]=E[0];$$
$$A[1]+B[1]=C[1]; \qquad C[1]+D[1]=E[1];$$
$$\cdots\cdots$$
$$A[n]+B[n]=C[n]; \qquad C[n]+D[n]=E[n].$$

The average number of clock cycles per addition will be still 6. However, if we perform the two vector additions in the following sequence:

$$A[0]+B[0]=C[0];\ A[1]+B[1]=C[1];\ \ldots\ldots;\ A[n]+B[n]=C[n];$$
$$C[0]+D[0]=E[0];\ C[1]+D[1]=E[1];\ \ldots\ldots;\ C[n]+D[n]=E[n].$$

The average number of clock cycles per addition will be reduced to $2+4/(2n)$. This is because the neighboring instructions are independent from each other and no data hazard need to be avoided by adding bubbles. The $4/(2n)$ term comes from 4 bubbles

inserted into the pipeline to avoid structure hazards. Recalling the two rules that we drew for designing a structure-hazard-free instruction set, we will understand that vectors C and D have to be written to a memory bank other than the memory bank in which vectors A, B and E are stored, so that the pipeline will read from one memory bank and write to another memory bank thus avoiding write conflicts. However, when we are about to finish the first vector additions and start the second vector additions, we have to wait until the last result of the first vector additions have been written back to memory, then we can start to perform the second vector additions. Hence 4 bubbles are added to avoid conflicts.

Up until now, instead of considering the average CPI of DSP-RAM, we have been considering the average number of clock cycles per operation (addition, for the case of our examples) for DSP-RAM. This is because most operations need more than one instruction to complete. Note that for different architectures, the numbers of instructions needed for completing the same calculation are usually different. Hence the clock cycles per calculation or operation may give a more accurate idea than CPI. To explain this clearly, let's compare DSP-RAM with a load-store architecture.

Most modern microprocessors have adopted the load-store or register-register architectures. They use load and store instructions to exchange date between memories and register files, and perform computation among their registers. If we need to compute A+B=C, generally we need to issue the following four instructions:

Load r0, (A); Load r1, (B); Add r2, r0, r1; Store r2, (C).

Here the (A), (B) and (C) stand for the references to variables A, B and C respectively, that are held in memory locations. The r0, r1 and r2 are three registers.

If variables A, B and C are already in the register file, which is usually the case for scalar computations, we can omit the load and store instructions and the overall number of instructions required to complete the addition is one. Compared to DSP-RAM, which generally needs two instructions to finish one addition, load-store architectures usually have greater performance at the same frequency. However, for vector computations, the load-store architectures need load and store instructions to read and write vector elements from/to memory if the vectors can not be completely stored in the register

file. Therefore we need four instructions to complete an addition on a load-store architecture. DSP-RAM's PE, however, needs only $2+4/(2n)$ clock cycles per addition (n stands for the vector length. The $4/(2n)$ term can be neglected when n is large), which is about 2 times faster than load-store architectures if they are operated at the same frequency.

Earlier we showed that the performance of an application developed on the DSP-RAM architecture is determined by the fraction of operations that can be accelerated, the speed up brought to these operations, and the portion of any extra operations introduced by data initialization and context switches. To increase the fraction of accelerateable operations, we need to increase the versatility of DSP-RAM's PEs. To further speed up the accelerateable fraction of operations, we should carefully design the processing flows and data flows to fully utilize DSP-RAM's computational power, while at the same time avoiding data hazards and structure hazards.

In the next section, we will discuss the programmability of DSP-RAM.

3.1.3 DSP-RAM's Programmability

To use microprocessor hardware to solve a substantial problem, we need to split up the problem into many smaller problems, and then find solutions for these small problems. These solutions usually comprise many basic operations that can be done on hardware. The behavior of the microprocessor hardware is controlled by machine code. It's often possible to manually write machine code to directly control and schedule hardware to realize those basic operations, and eventually solve the problem. However, because the control and scheduling of microprocessor hardware can be very complex, low-level programming is usually best avoided for all but the simplest problems.

To improve the programmability of microprocessors, low-level and high-level languages were invented to hide the hardware details from programmers. Programmers need to know the definition of languages, and based on this, they can compose operations into functions and programs. A compiler automatically translates programs into machine codes, and a linker will compose machine codes into a complete executable file.

To hide low-level software details, reusable functions and data structures are collected into libraries. Those functions needed by every computer, such as file operations, process management, communication control, are then assembled together to become operating systems. The operating system in turn provides APIs and system calls.

Generally, the fewer hardware and software details that need to be handled directly by a programmer, the more the programmer can concentrate his/her efforts on solving the problem itself. Hence, the programmability of microprocessor architecture is to a significant degree determined by how many hardware and software details it exposes to programmers.

DSP-RAM research is still in its very early stages. No compiler has been developed to support high-level or low-level languages on DSP-RAM. Therefore at present we can only program DSP-RAM at the machine-code level. Furthermore, because DSP-RAM's pipeline is not designed to operate as a black box and we have to take care of every details of pipeline control, programming DSP-RAM tends to be complicated.

As shown in **Figure 3.1-2**, to program a black-box pipeline, an instruction only needs to deal with the data issued with it, and once the data and instruction are issued to the pipeline, it will automatically flow through the pipeline. For the current version of DSP-RAM, an instruction has to consider all of the data currently in the pipeline; therefore the neighboring instructions have to be correlated with each other to process data correctly.

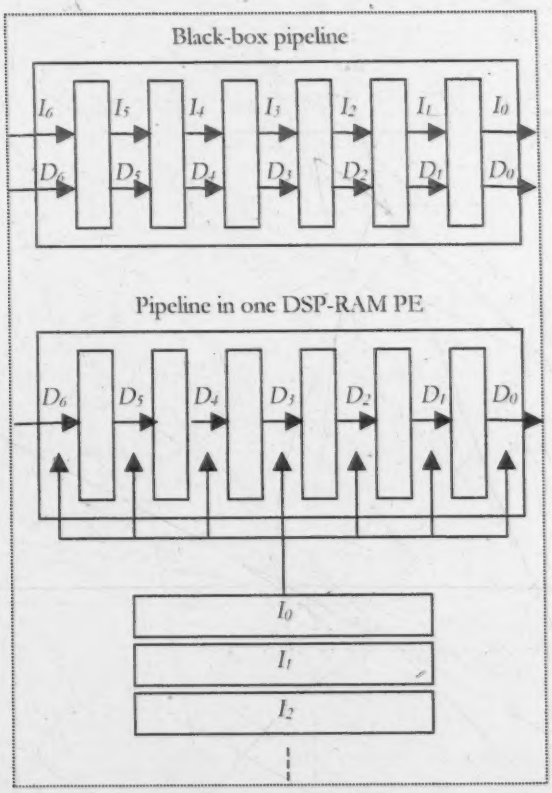

Figure 3.1-2 Comparison between DSP-RAM's pipeline and a black-box pipeline

There are many potential approaches to improve the programmability of the current DSP-RAM architecture. One approach is to make the pipeline a black-box pipeline, as shown in **Figure 3.1-3**. The pipeline registers are lengthened to be able to store and propagate the control signals associated with each set of input operands. In this way, the instruction stream and the data stream are synchronized. Every instruction is decorrelated with its neighboring instructions and only needs to concern itself with operands associated with each instructions. Programming the modified architecture would be similar to programming other microprocessor architectures; hence a programmer would need less training to program on the modified architecture than on current architecture. In addition, because of the functional simplicity of every instruction, this approach would also simplify the design of instruction set and the future possible compiler design. The main downside of this approach is that, because it

needs more registers at every stage to store control information, it increases the transistor budget and the complexity of the PE. This tends to reduce the number of PEs that can be integrated on a chip.

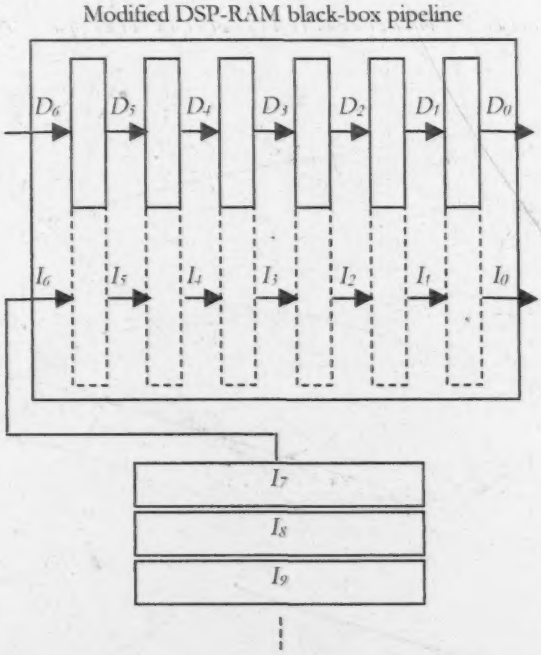

Figure 3.1-3 Modified DSP-RAM black-box pipeline

We considered another approach to simplify DSP-RAM programming. This approach does not require changes to the current architecture, but still helps a lot when programming solutions to complex problems such as the DCT and DWT on DSP-RAM.

According to our experience, the main difficulty when programming the current version of DSP-RAM is the synchronization of the instruction stream and data streams, especially the addressing problem for data streams. In other words, it is hard to figure out when the pipeline should perform what operation on which data sources, and when the result should be output to where. The timing in DSP-RAM programming is much more important because an instruction has to take care of the data that has been input into the pipeline several clock cycles before. A programmer has to write instructions to

perform different operations on the data in different stages, and it's very easy to get lost in the complex timing relationships.

Our approach uses three state machines to control input, output and operation of the pipeline, respectively. When we design the input and output state machines, we only need to concentrate on the input and output patterns of the target algorithm. And when we design the control state machine, we will only concentrate on the operation of the pipeline. After that, we will concentrate on the synchronization of three state machines. In this way, we divide the whole programming process into several stages and simplify the task of every stage.

Let's use the following generalized vector computation as an example: A, B, and C are vectors of constants, where X, Y, and Z are vectors of variables. The indexes for these vectors are different so that these vectors can be accessed in different patterns.

$$Z[i] = (A[m] \times X[j]) + (B[n] \times Y[k]) + C[l] \quad \text{Equation 3.1-4}$$

Assume that these vectors are all very long so that we need to split them up into multiple equal-sized smaller pieces and assign each PE one piece. To perform this vector computation on PEs, we will broadcast the constant vectors A, B, C to the PEs because such constant coefficients are generally needed by all the PEs. We will store the variable vectors in the local memory of PEs. Especially, we put inputs X and Y into a memory bank and put output Z into the other memory bank so as to avoid structure hazards.

First let's design the control state machine. Neglecting the address or index calculating problem, we will have pipeline control flow shown in **Table 3.1-4**.

Table 3.1-4 Pipeline control flow for the vector computation AX+BY+C

Clock Cycles	Pipeline stages						State
	BB	M0	M1	M2	ACC	WB	
CC0	A						State 1
CC1	B	X×A				Bubble	State 2
CC2	C	Y×B	X×A				State 3
CC3	A	C×1	Y×B	X×A			State 1
CC4	B	X×A	C×1	Y×B	AX	Bubble	State 2
CC5	C	Y×B	X×A	C×1	AX+BY		State 3
CC6	A	C×1	Y×B	X×A	AX+BY+C		**State 1**
CC7	B	X×A	C×1	Y×B	AX	Z	**State 2**
CC8	C	Y×B	X×A	C×1	AX+BY		**State 3**
CC9	Bubble	C×1	Y×B	X×A	AX+BY+C		State 1
CC10	Bubble	Bubble	C×1	Y×B	AX	Z	State 2
CC11	Bubble	Bubble		C×1	AX+BY		State 3
CC12	Bubble	Bubble			AX+BY+C		State 1
CC13	Bubble	Bubble				Z	State 2
CC14	Bubble	Bubble					State 3

There are only three pipeline control states in the process and every three control states comprise a processing cycle. The pipeline will repeat the processing cycle until all the Z values are outputted. In state 1, as shown completely in CC6, the BB stage needs to input the next element from vector A from the broadcast bus to load it into broadcast bus register. The M0 stage needs to input a stored constant value of "1" from the input memory bank and needs to multiply it with the value that was loaded into broadcast bus register. M1 and M2 are inside the multiplier and do not require any extra control. The ACC stage will accumulate and get the result of AX+BY+C for the previous processing cycle, and the WB stage will do nothing. The state 2 and state 3, as shown completely in CC7 and CC8, do similar things except that State 2 outputs the Z value for the previous processing cycle in the WB stage.

We can describe these behaviors with the control state machine shown in **Figure 3.1-4**.

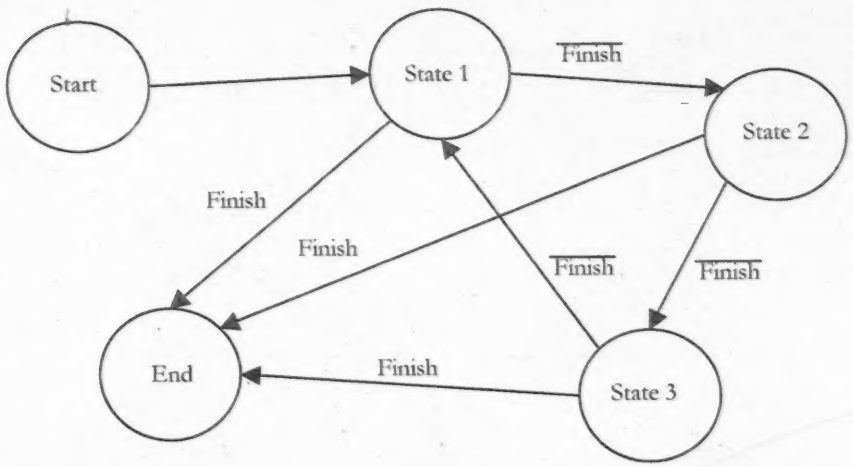

Figure 3.1-4 Control State Machine for vector computation AX+BY+C

In the states 1, 2, and 3, instructions are issued to perform the required operations in the pipeline as described in **Table 3.1-4**. We will also check if all the Z values have been outputted. If this is the case, then the control state will transit to the state "End".

From **Table 3.1-4** we see that the Z values in the first two processing cycles, i.e. in CC1 and CC4, are invalid result because of the start-up latency of the pipeline. In the last two cycles, shown as CC9-CC14 in **Table 3.1-4**, the further inputs are neglected because they are not going to be involved in any calculations.

We can also easily design our input and output state machines with the information in **Table 3.1-4**. The indexes of all the vectors will be updated for every processing cycle, except for the output state machine in the first two processing cycles and for the input state machine in the last two processing cycles. The address update can be very varied depending on the target algorithms. **Figure 3.1-5** and **Figure 3.1-6** show the state transition diagram of the input and output state machines for the vector computation AX+BY+C.

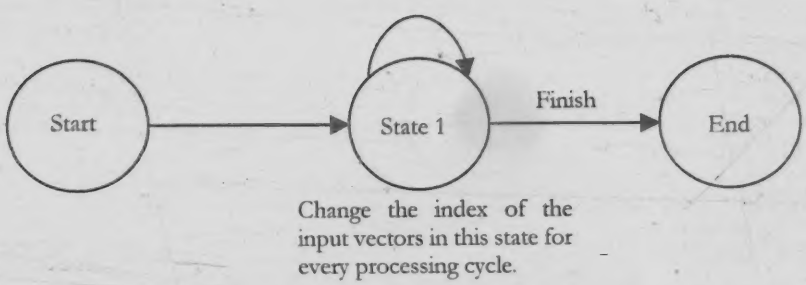

Figure 3.1-5 Input state machine for the vector computation AX+BY+C

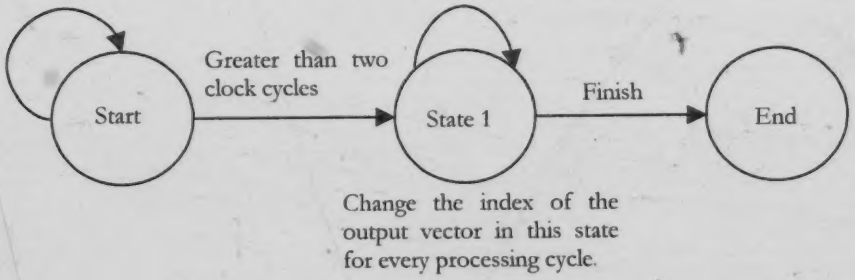
Start
Greater than two clock cycles
State 1
Finish
End
Change the index of the output vector in this state for every processing cycle.

Figure 3.1-6 Output state machine for the vector computation AX+BY+C

Table 3.1-4 also shows that DSP-RAM's PE can finish three multiplications and two additions in 3 clock cycles, i.e. 0.6 clock cycles per operation, if the first five clock cycles that are needed to fill up the pipeline are neglected.

The approach of using state machines to control the behavior of a pipeline also prompts an interesting idea for host controller design and compiler design. Instead of writing machine codes to program DSP-RAM, we may configure the state machines and use them to generate the machine instructions at run time.

In this section, we discussed challenges faced by the DSP-RAM architecture when dealing with system-level problems. We then proposed and analyzed ideas and methodologies to solve these challenges. In the next section, we will put all these ideas and methodologies into practice as we implement a JPEG encoder on DSP-RAM.

3.2 JPEG Implementation on DSP-RAM

In this section, we describe an implementation of a JPEG encoder on DSP-RAM, which can compress any RGB bitmap image into a JPEG image file. Because a DSP-RAM chip and development kits are not yet available, our work is based on a high-level DSP-RAM simulator written in C++. In Chapter 4, we will introduce an FPGA implementation of DSP-RAM. Based on the information from this section and in Chapter 4, we will evaluate the performance of our parallel JPEG encoder in Chapter 5.

As we introduced in Section 2.1.2, the architecture of our targeted JPEG encoder is given in **Figure 3.2-1**.

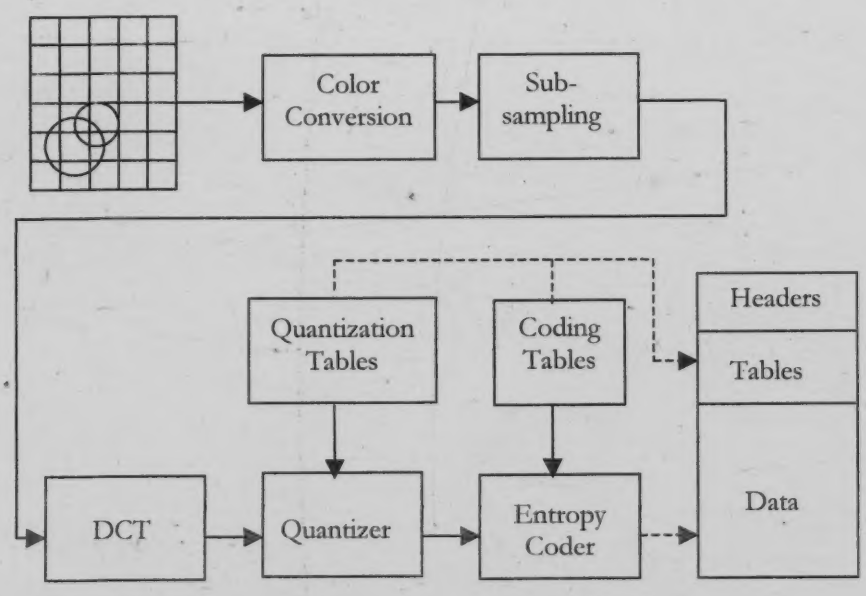

Figure 3.2-1 Block diagram of our targeted JPEG encoder

In Section 3.2.1, we will describe how the image data is initialized and organized for processing. In Section 3.2.2, we will introduce how functions are assigned to the DSP-RAM and the host controller. From Section 3.2.3 to Section 3.2.7, we will introduce the implementation of all the functional blocks in detail.

3.2.1 Data Initialization and Organization

Before we start to process data on DSP-RAM, we need to determine what data should be loaded into DSP-RAM's local memory, and how these data should be organized. These questions not only affect how we process the data, they also affect the memory requirements of the final implementation and the overhead caused by initialization and reorganization.

In JPEG compression, there are two types of data. One type is constant (or only slowly changing) data, such as the data in quantization tables and entropy coding tables. They are generally pre-calculated and are small in size (less than 1K). Another type of data are the variables, such as pixels in the spatial domain and coefficients in the frequency domain. They are the objects that need to be processed and are generally large in size (can be several Mbytes).

As we can see from DSP-RAM's architecture, the PEs input data either from the broadcast bus, the shift bus or from their local memory. Naturally, we would rather put variables in local memory to utilize DSP-RAM's great internal memory bandwidth. As for the constants, because they are needed by all the PEs, it is natural to distribute them over the broadcast bus.

How should the variables be organized in local memory? In other words, how should the variables be distributed to the local memory of the PEs? And how should data be transferred during processing?

We should keep in mind that data transfers to and from the DSP-RAM will increase the overhead and should be minimized. Second, too many variables in a PE's local memory will increase the memory requirements of the DSP-RAM architecture, while too few variables in a PE's local memory may be not enough for the PE to finish all the processing, and hence may cause extra data transfers during processing.

Let's take a look at how an image is compressed.

In color conversion functional blocks, the original image is transformed from the RGB representation to the YCbCr representation. The computation is performed pixel-by-pixel. Hence the Minimum Coded Unit (MCU) that can be processed in parallel is the pixel. In Subsampling functional blocks, the pixels in the Cb and Cr planes

are downsampled by computing the average value of four neighboring pixels as shown in **Figure 2.1-5**. The MCU at this stage is a 2×2 pixel block. In the DCT functional blocks, the DCT is computed block-by-block and the MCU is an 8×8 pixel block. In the Quantization functional block, each of the DCT coefficients is divided by a quantization coefficient according to its position in the 8×8 block, so the MCU is still an 8×8 coefficient block. In the Entropy Encoding stage, the encoding is also based on 8×8 blocks, so the MCU is still an 8×8 block of data.

From the process described above, we can understand that the MCU for the entire process is three 16×16 blocks in the RGB planes. This means that the minimum amount of data needed to be stored in a PE's local memory for completing the entire compression process, without the need to transfer data from other PEs, is three 16×16 blocks from the same position in the RGB plane. Note that, although the 8×8 block is the largest MCU for individual functions, the information stored in an 8×8 block at the output of the process is originally stored in a 16×16 block before downsampling, and any pixel in any color planes after Color Conversion processing actually contains information from the corresponding pixels from the entire RGB plane.

In our implementation, we store one MCU of data per PE, and every PE works independently on its own data. In this way, all the processing is local so no intercommunication is needed among functional blocks and no data reorganization is needed when proceeding to a new function. The memory requirement for every PE is thereby kept to less than 4K Bytes.

3.2.2 Processing Pattern

When we compress an image, the source image may be stored in a hard disk, or be transmitted from a communication channel. Whatever it is, it will be divided into many 16×16 blocks. If the blocks at the edge of the image are not filled up, the JPEG encoder will extend the source image with the pixel values at the edge to fill up those blocks. Then we group every three blocks at the same position of the RGB plane into a MCU and store them directly into DSP-RAM's local memories, one MCU per PE.

A MCU contains 16×16 pixels in total, and every pixel will have three samples in the RGB plane. So the overall number of samples is 768. In this thesis, we will only deal with samples with a bit length of 8.

We put each MCU into a PE's bank A, and depending on how many PEs we have in the DSP-RAM architecture, we will process MCUs in parallel for the functional blocks in the JPEG encoding flow until we reach the entropy encoding function.

All of the samples are stored in one memory bank. When we process data in a PE, we always input data from one memory bank and the broadcast bus, and output the result to the other memory bank. Because data always flows from one memory bank to the other memory bank, the structure hazard that exists in DSP-RAM's pipeline will not occur. However, when we need to change the data flow direction, we have to add bubbles into the pipeline to avoid structure hazards. For this reason, we will always try to minimize the times that we need to change the data flow direction in our algorithms.

The two directions of data flow are shown as **Figure 3.2-2**. Throughout the parallel processing part of the JPEG implementation, we will alternate between pass 1 and pass 2 until we will finish all the functions that can be done efficiently on the DSP-RAM architecture.

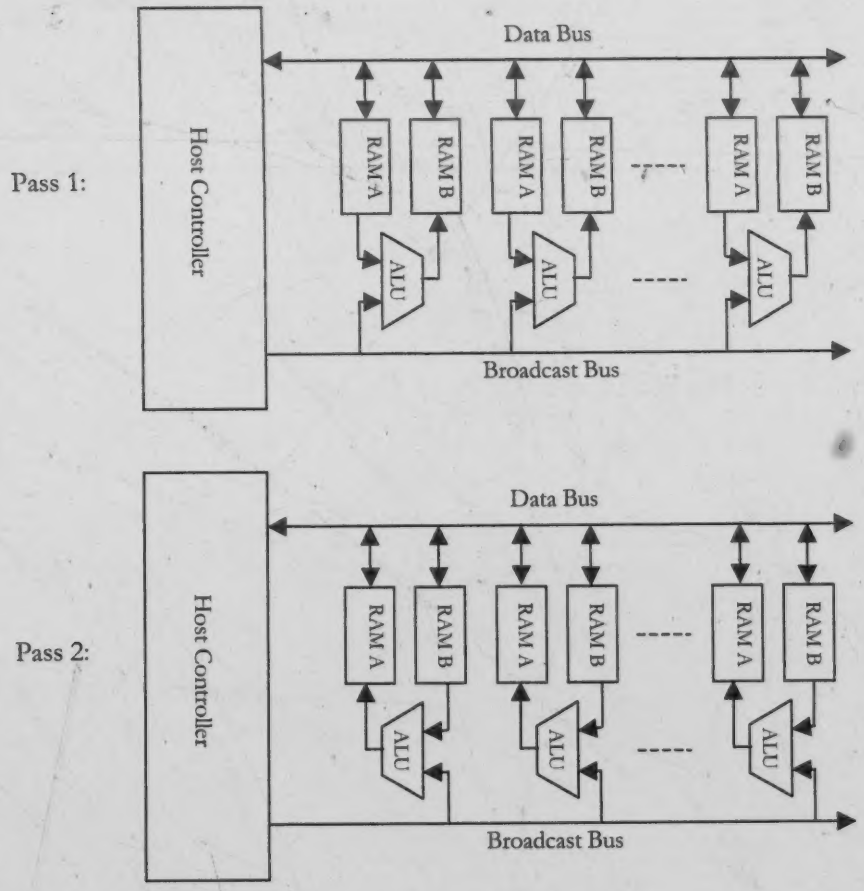

Figure 3.2-2 Data flow in the DSP-RAM implementation

3.2.3 Implementation of Color Conversion

As we introduced in the Color Conversion discussion in section of 2.1.2, we need to calculate a pixel's YCbCr values according to its RGB values with the following equations:

$$Y = (0.29900 \times R) + (0.58700 \times G) + (0.11400 \times B) \qquad \text{Equation 3.2-1}$$

$$C_b = -(0.16874 \times R) - (0.33126 \times G) + (0.50000 \times B) + 128 \qquad \text{Equation 3.2-2}$$

$$C_r = (0.50000 \times R) - (0.41869 \times G) - (0.08131 \times B) + 128 \qquad \text{Equation 3.2-3}$$

Recall the example that we gave in Section 3.1.3, and note that these new equations are very similar to Equation 3.1-4. We can thus perform the calculation as we described in

Section 3.1.3. However, because the coefficients in these equations are very small and DSP-RAM's ALU does not support floating-point calculation, some precision may be lost due to underflow or overflow if we directly apply these equations blindly.

To solve the precision problem, we multiply or scale all the constants in the equations by a factor of 128, i.e. left shift them by 7 bits, before we broadcast them into DSP-RAM. After we finish the calculations, we right shift the results back by 7 bits. In this way we can utilize the width of our data path to achieve a reasonable precision. In addition, before we right shift the results, we add a 64 to the result if it is positive or deduct a 64 to the result if it is negative, to round up the final result to the nearest integer [32].

The color conversion equations are generalized as follows:

$$Y = (C_{RY} \times R) + (C_{GY} \times G) + (C_{BY} \times B) + C_Y \qquad \text{Equation 3.2-4}$$

$$C_b = (C_{RCb} \times R) + (C_{GCb} \times G) + (C_{BCb} \times B) + C_{Cb} \qquad \text{Equation 3.2-5}$$

$$C_r = (C_{RCr} \times R) + (C_{GCr} \times G) + (C_{BCr} \times B) + C_{Cr} \qquad \text{Equation 3.2-6}$$

These equations can be further abstracted as shown in Equation 3.2-7.

$$Q = (A \times X) + (B \times Y) + (C \times Z) + D \qquad \text{Equation 3.2-7}$$

This equation has a form similar to Equation 3.1-4, hence its pipeline control flow is similar to that shown in **Table 3.1-4** except that it takes four clock cycles on average to get the output Q.

To perform color conversion on a MCU, we need to perform the calculation of Equation 3.2-7 768 (i.e. 3×16×16) times. If we include the five clock cycles that are needed to avoid hazards, then the overall number of clock cycles will be 3077 (4×768+5).

To perform Equation 3.2-7 on a non-parallel load-store superscalar architecture, and suppose that the constant coefficients are only loaded once and stored in registers, then the overall number of instructions needed is 10 (3 loads, 3 multiplications, 3 additions and 1 store). Therefore the overall number of instructions needed for processing a MCU is 7691 (10×768+11, the 11 is for loading 12 constant coefficients, but because

C_{Cb} and C_{Cr} are identical, we only need 11 loads for 12 coefficients). Suppose the CPI for the superscalar architecture is one, then we would still need 7691 clock cycles to perform color conversion for a MCU. This value is 2.25 times greater than the number we get for the implementation on DSP-RAM.

However, the biggest advantage of DSP-RAM comes from its ability to process MCUs in parallel. When the number of MCUs increases, the number of clock cycles needed to perform color conversion on them for DSP-RAM will not increase linearly as it does for a non-parallel architecture; instead, it will stay the same until the number of MCUs is greater than the number of processing units. **Figure 3.2-3** shows such a comparison between DSP-RAM with three PEs and a non-parallel architecture. The horizontal axis stands for the number of MCUs needed to be processed. The vertical axis stands for the number of clock cycles. The unit for vertical axis is 3077 clock cycles.

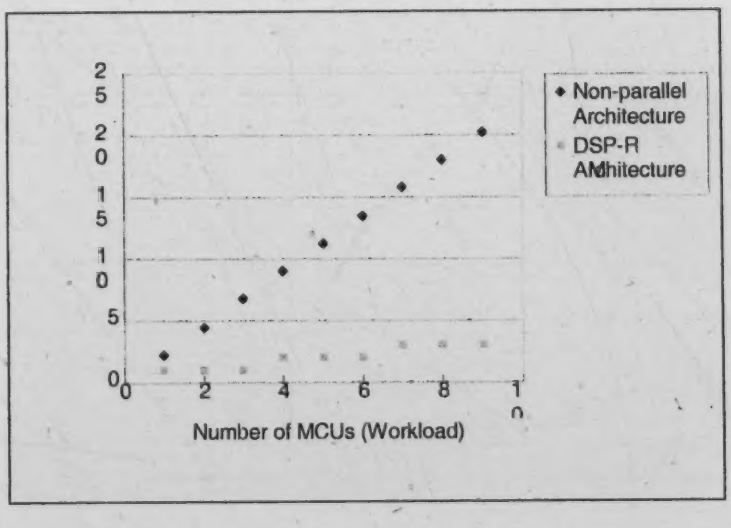

Figure 3.2-3 Comparison of clock cycles needed for DSP-RAM and a non-parallel architecture to perform color conversion function

3.2.4 Implementation of Subsampling

In our JPEG encoder, we subsample the pixels in the Cb and Cr planes by a factor of 2. The basic calculation can be expressed as in Equation 3.2-8.

$$P'[i][j] = \frac{P[2i][2j] + P[2i+1][2j] + P[2i][2j+1] + P[2i+1][2j+1]}{4} - 128$$

where $i, j \in [0,7]$ Equation 3.2-8

In our implementation, we substitute the division by 4 with a right shift by 2 bits. To reduce the quantization noise introduced by averaging and downsampling, a common DSP technology called dithering is used here by adding a small amount of noise to keep more information of original data [32]. The modified algorithm is shown as Equation 3.2-9.

$$P'[i][j] = \frac{P[2i]\,[2j] + P[2i+1]\,[2j] + P[2i]\,[2j+1] + P[2i+1]\,[2j+1] + Alias}{4} - 128$$

$$i, j \in [0,7],\ Alias = 1,2,1,2,... \qquad \text{Equation 3.2-9}$$

Equation 3.2-9 can be generalized as Equation 3.2-10,

$$Q = (X + Y + Z + W + A) >> 2 \qquad \text{Equation 3.2-10}$$

where X, Y, Z, W and Q are variables, and A is a constant calculated from the Alias term and the -128 term in Equation 3.2-9. The operator ">>" stands for right-shift.

Because the right shift can be done with an accumulation at the same clock cycle on DSP-RAM, Equation 3.2-10 has a form similar to Equation 3.1-4. Hence its pipeline control flow is also similar to that shown in **Table 3.1-4** except that it takes five clock cycles to get the output Q instead of three clock cycles.

We need to perform the calculation of Equation 3.2-10 128 (8×8×2) times to downsample the two 16×16 blocks in the Cb and Cr planes, and so the overall number of clock cycles is 640 (5×128). After the downsampling process, the data in the Cb and Cr planes will have been compressed by a factor of four and they will be stored back in bank A.

We also need to move the Y plane pixels from bank B to bank A so that all the effective data will be in the same memory bank when we start to process the next function. This is an overhead resulting from the DSP-RAM architecture since there is no need to do this for load-store architecture. However, we can utilize this extra process to change the data structure from a 16×16 block into four 8×8 blocks to simplify data addressing for

the following DCT functions. The overhead introduced by the data movement at this stage is 256 (16×16) clock cycles.

Because the direction of data movement is the same as in the subsampling processing, there is no need to add bubbles between them. However, we still need to add 5 bubbles at the end of the processing to avoid potential hazards caused by the following function. Therefore, the overall number of clock cycles required by our implementation to perform the subsampling function on a MCU is 901 (640+256+5).

To perform Equation 3.2-10 on a non-parallel load-store architecture, supposing we store the constants in registers, we will generally need 10 instructions (4 loads, 4 additions, 1 shift and 1 store). Therefore the overall number of instructions needed to perform subsampling on a MCU is 1283 (10×128+3, the 3 is for loading constants).

3.2.5 Implementation of DCT

As we introduced in Section 2.1.2, there are many algorithms available for DCT processing. To choose a suitable algorithm for the DSP-RAM architecture, we need to understand the characteristics of the DSP-RAM architecture. First, DSP-RAM achieves its high efficiency when it is performing vector computation. This is because vector computations generally require fewer changes in the data flow direction, and hence require fewer bubbles to be inserted into the pipeline to avoid possible hazards. Second, multiplication and addition have the same latency on DSP-RAM, therefore, those algorithms strive to reduce the number of multiplications in favor of additions may not necessarily be more efficient on DSP-RAM.

In our implementation, we chose the DCT algorithm described in **Figure 2.1-7** and Equation 2.1-7 for the sake of efficiency and to simplify memory management. Equation 2.1-7 can be generalized as in Equation 3.2-11.

$$Q = (A[0] \times P[0]) + (A[1] \times P[1]) + ... + (A[7] \times P[7]) \qquad \text{Equation 3.2-11}$$

where Q stands for the coefficients in the frequency domain, P[0]~P[7] stands for a row or column of pixels in a 8×8 block in the spatial domain, and A[0]~A[7] stands for the pre-calculated constant coefficients in Equation 2.1-7. Equation 3.2-11 still has a

form similar to Equation 3.1-4. Its pipeline control flows is also similar to that in **Table 3.1-4** except that it takes 8 clock cycles to get the output Q.

As we discussed in **Figure 2.1-7**, to perform a FDCT on an 8×8 pixel block, we first perform a 1-D DCT according to Equation 3.2-11 going through all the eight rows in one memory bank, and leaving intermediate results in the form of a 8×8 data block stored in the other memory bank Then we perform a 1-D DCT with Equation 3.2-11 going through all the eight columns of the intermediate data block, and the original pixels block will be overwritten by the final results.

To perform the FDCT on an MCU, which contains six 8×8 blocks at the current stage, we will perform a 1-D DCT over all the rows of all the six blocks before we begin to process columns. In this way, there is only one direction change of data flow so that only 10 bubbles need to be added to avoid hazards.

Therefore, we need to perform the calculation of Equation 3.2-11 768 (8×8×2×6) times to perform the DCT on an MCU in DSP-RAM, and the overall number of clock cycles needed is 6154 (768×8+10).

Here we use the widely-used JPEG encoder from IJG as an example for comparison [32]. IJG implemented their 1-D DCT function based on an algorithm with 12 multiplications and 32 additions. If all the intermediate results and constant coefficients are stored in registers, then the instructions needed to perform the DCT on a row/column of 8 pixels is 60 (12 multiplications, 32 additions, 8 loads and 8 stores). Therefore, the overall number of instructions for a MCU will be 5760 (60×8×2×6).

Because we chose the basic 1-D DCT algorithm, our implementation needs 64 clock cycles to perform Equation 3.2-11 on a row/column of pixels using DSP-RAM. If IJG's implementation runs on an architecture with an CPI of 1, it will be 4 clock cycles faster than our approach to processing one row/column of data. But if we implement the DCT algorithm adopted by IJG on DSP-RAM, the number of clock cycles needed to process a row/column of data will be greater than 65 due to the bubbles added to avoid hazards. This is why we choose to use the simple basic 1-D DCT algorithm on DSP-RAM.

To retain numerical precision during the calculation, we used a method similar to the one that we used in the process of color conversion. We scale the input numbers to fully utilize the available width of our data path, and then rescale the results back at the end of the calculation before we write them back into memory.

3.2.6 Implementation of Quantization

To discard the less important information in the coefficient blocks after the DCT process, we quantize these coefficients by dividing them by a certain number determined by their position in the 8×8 blocks. The divisors are calculated from the tables that are recommended in the JPEG standard. The process of quantization can be expressed as Equation 3.2-12.

$$Q[i][j] = \frac{P[i][j]}{C[i][j]} \qquad i, j = 0,1,...7 \qquad \text{Equation 3.2-12}$$

where the P[i][j] and Q[i][j] stand for coefficients before and after the quantization process, and the C[i][j] stands for the constant divisors pre-calculated from quantization tables.

To round up the results to the closest integer, we will add a C[i][j]/2 term to the dividend when the dividend is positive or subtract a C[i][j]/2 term when it is negative. Also, because DSP-RAM's ALU does not support division directly, we substitute the division by constants with a multiplication by the reciprocals of the constants. The modified equation is shown as Equation 3.2-13,

$$Q[i][j] = \left(P[i][j] \pm \frac{C[i][j]}{2} \right) \times \frac{1}{C[i][j]} \quad i, j = 0,1,...7 \quad \text{Equation 3.2-13}$$

This can be abstracted as in Equation 3.2-14.

$$Q = (P + X \times A) \times B \qquad \text{Equation 3.2-14}$$

where P and Q stand for the coefficients before and after the quantization process, and X is an array of variables whose value is 1 or -1 depending on the value of the

coefficients in the same position of P. A and B are constant arrays that are pre-calculated from the C[i][j] in Equation 3.2-13.

To determine the variables in X, we allocate a block of memory with the same size as a MCU in the same memory bank as where frequency coefficients are stored; these variables are initialized with a value 1. The frequency coefficients are later read from memory one-by-one, compared with zero, and the result is used to control the memory write enable when we write a -1 to the variables in X. For every variable in X, we need 4 clock cycles to determine its value, and the overall number of clock cycles to determine the entire array of X is 1536 (4×8×8×6). This is also an overhead brought by the DSP-RAM architecture. We would not have to do this when we perform quantization on a general-purpose microprocessor.

To evaluate Equation 3.2-14 on DSP-RAM, we must first calculate $(P + X \times A)$ and then store the intermediate result into the memory bank other than the one we input P and X. Then we multiply the intermediate result with B and overwrite the original coefficients with the products. The first step needs 2 clock cycles per coefficient, the second step needs one clock cycle per coefficient, and we need to add 5 bubbles between these two steps to avoid hazards.

Therefore the overall number of clock cycles to perform quantization on a MCU (six 8×8 pixel blocks) on DSP-RAM is 2693 (1536+3×8×8×6+5).

If we evaluate Equation 3.2-14 on a non-parallel load-store architecture, we would need only 2 instructions (1 branch and 1 assignment) to determine a variable in X, and therefore needs 8 instructions (2 loads, 2 for X, 1 addition, 2 multiplications and 1 store) to quantize one DCT coefficient. So the overall number of instructions for a MCU is 3072 (8×64×6).

3.2.7 Entropy Encoder and File Format Processing

As we introduced in Section 2.1.2, entropy encoding involves many shift operations, bit operations and table look-up operations. Some of them are impossible, and the rest are at least inefficient, when implemented on the current DSP-RAM. Although it's possible to process this part in parallel by enhancing the versatility of DSP-RAM's ALU, we

chose instead to perform this processing sequentially on the host controller due to time constraints.

After entropy encoding, the original image is compressed into a bit stream. JPEG also defines a set of file formats to contain the compressed data and other information. Hence, a JPEG encoder needs to generate the final JPEG file for storage and transmission. This part of the processing was also assigned to the host controller because it involves many non-parallelizable interactions with the other parts of the system.

In our JPEG encoder, we modified the codes from IJG for these two parts and embedded them into our codes. This approach also simplifies our work when we compared our JPEG encoder with IJG's encoder because of the similarity of the two implementations.

3.3 DWT Implementation on DSP-RAM

Because of time constraints, we did not implement a true JPEG 2000 encoder on DSP-RAM. Instead, we implemented a block-based DWT functional block as defined in JPEG 2000 and embedded it into our completed JPEG encoder processing flow.

3.3.1 DWT Processing in JPEG 2000

According to the definition of JPEG 2000 [38], multi-level DWT processing of an 8×8 pixel block can be expressed as shown in **Figure 3.3-1**.

In **Figure 3.3-1**, we use two-letter labels (LL, LH, HL, HH) to indicate which sub-band within a wavelet processing level the block of coefficients are in. For example, a block labeled LH contains coefficients that are in the low-pass band in horizontal direction and in the high-pass band in vertical direction. In the beginning (level 0), all the pixels in the input block are considered to be in a LL sub-band. For every level, the 2-D DWT algorithm will only process the data from the LL sub-band from the previous level and will replace the LL block with four smaller blocks that are one quarter as big. For a full decomposition the algorithm will repeat the process until there is only single pixel in the resulting LL block. In the case of performing DWT on an 8×8 pixel block, a full decomposition takes 3 levels of processing, as shown in **Figure 3.3-1**. Since the processes at different level are almost identical except for the size of input block, we

only implemented the first level of the processing on DSP-RAM architecture for our evaluation.

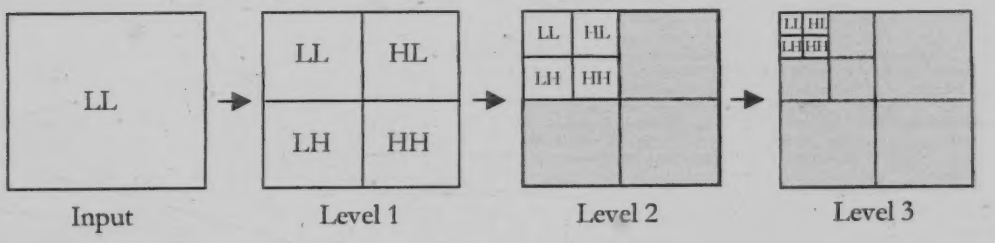

Figure 3.3-1 Top diagram of multi-level DWT processing of an 8×8 pixel block

The processing flow of our one-level 2-D DWT function is illustrated in **Figure 3.3-2**. We first perform a 1-D DWT on each row of pixel data, and then we perform a 1-D DWT on each column of coefficients. In the end we use a de-interleaving process to divide the block into four identical blocks (LL, LH, HL and HH) so that the coefficients are ready for the next level of processing.

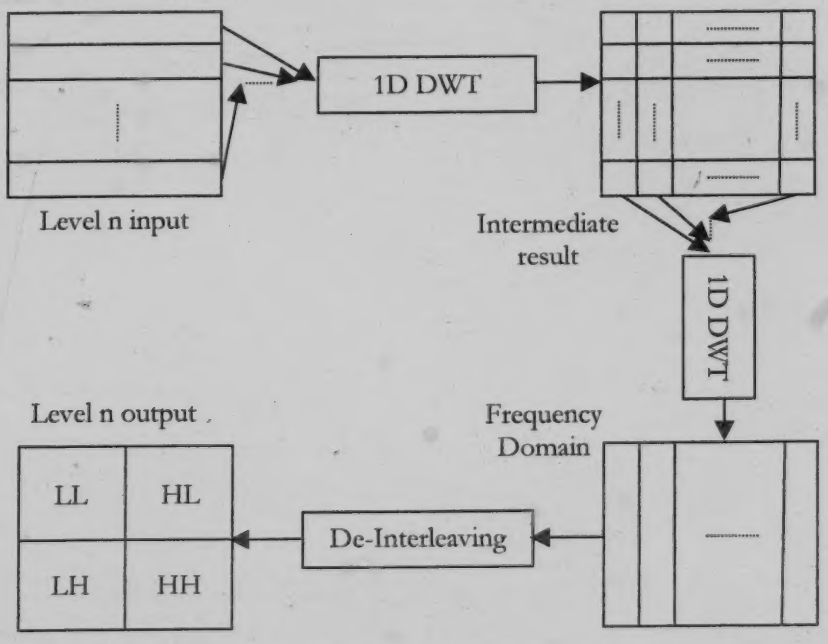

Figure 3.3-2 Using a 1-D DWT to perform a 2-D DWT transformation

The 1-D DWT coding architecture of JPEG 2000 was introduced in Section 2.1.3 and **Figure 2.1-10**. For the selection of the high-pass filter g[n] and the low-pass filter h[n], JPEG 2000 recommend two discrete wavelet filters: a reversible transformation and an irreversible transformation. There is no significant difference between the two transformations from an implementation point of view, so we chose the reversible transformation to construct the low-pass and high-pass filters for our DWT coding architecture.

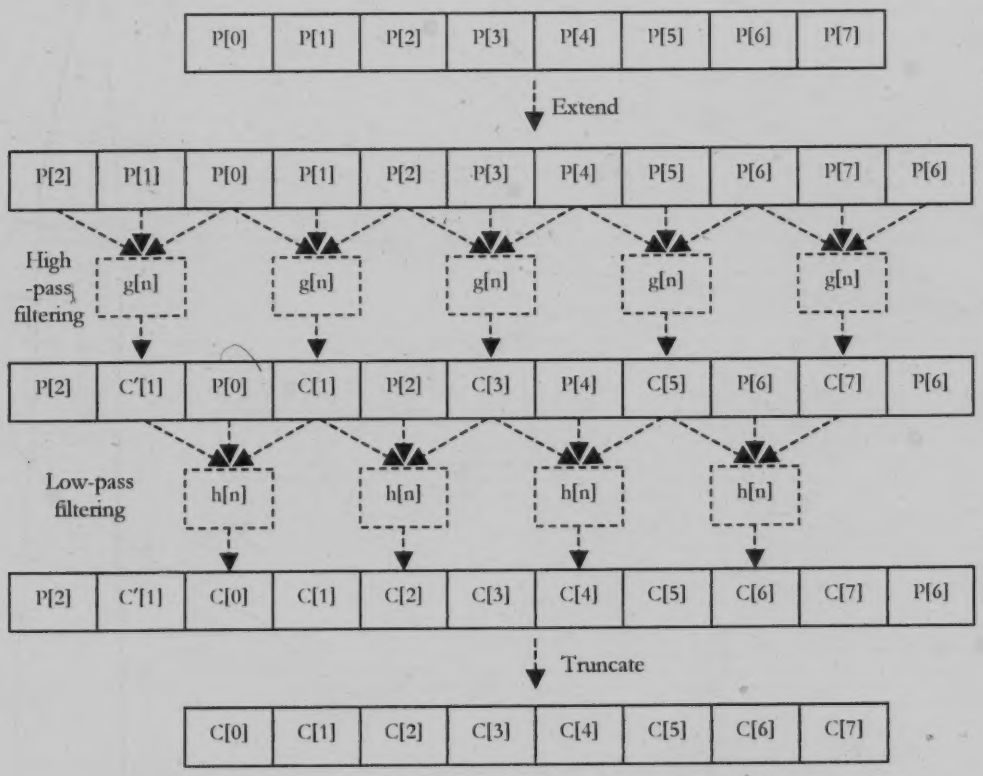
P[0] P[1] P[2] P[3] P[4] P[5] P[6] P[7]
Extend
P[2] P[1] P[0] P[1] P[2] P[3] P[4] P[5] P[6] P[7] P[6]
High -pass filtering
g[n]
P[2] C'[1] P[0] C[1] P[2] C[3] P[4] C[5] P[6] C[7] P[6]
Low-pass filtering
h[n]
P[2] C'[1] C[0] C[1] C[2] C[3] C[4] C[5] C[6] C[7] P[6]
Truncate
C[0] C[1] C[2] C[3] C[4] C[5] C[6] C[7]

Figure 3.3-3 Use of a reversible transformation to construct 1-D DWT filters

Figure 3.3-3 shows how the DWT filters are constructed with the reversible transformation according to JPEG 2000 [38]. First the input 1-D array is extended to left by two elements and to right by one to avoid sudden changes at the two boundaries. Then the extended array is passed through high-pass filters and all odd elements of the array are replaced with high-frequency coefficients. Then the resulting array is passed through low-pass filters and the even elements (except the two even elements at two

boundaries) are replaced with low-frequency coefficients. The process is ended by truncating the array back to its original size. Note that at this point, the high-frequency and low-frequency coefficients are now interleaved with each other; this is why we need the de-interleaving process shown in **Figure 3.3-2**.

The operation of the two filters g[n] and h[n] are defined in Equation 3.3-1.

$$C[2n+1] = P_{ext}[2n+1] - \left\lfloor \frac{P_{ext}[2n] + P_{ext}[2n+2]}{2} \right\rfloor, \text{ for } 2n+1 \in [i_0 - 1, i_l + 1).$$

$$C_{ext}[2n] = P_{ext}[2n] + \left\lfloor \frac{C[2n-1] + C[2n+1] + 2}{4} \right\rfloor, \text{ for } 2n \in [i_0, i_l).$$

Equation 3.3-1

In Equation 3.3-1, i_0 denotes the position of the first element of the original array in the extended array, and i_l denotes the position of the last element of the original array in the extended array. For the example in **Figure 3.3-3**, i_0=2 and i_l=9. The notation $\lfloor \rfloor$ denotes the floor function. The floor of an expression is the largest integer not exceeding the value of the enclosed expression.

3.3.2 Data Flow of the DSP-RAM Implementation of DWT

To make our DSP-RAM implementation more scalable, we still choose to utilize DSP-RAM's parallel architecture to exploit the parallelism of the outer loop of the DWT process. Therefore, we will still use one PE to perform all the DWT operations needed for one MCU. To efficiently implement the processing flow introduced in Section 3.3.1 on DSP-RAM, we divided the whole process into 12 passes according to different data flow patterns that are shown in **Figure 3.3-4**.

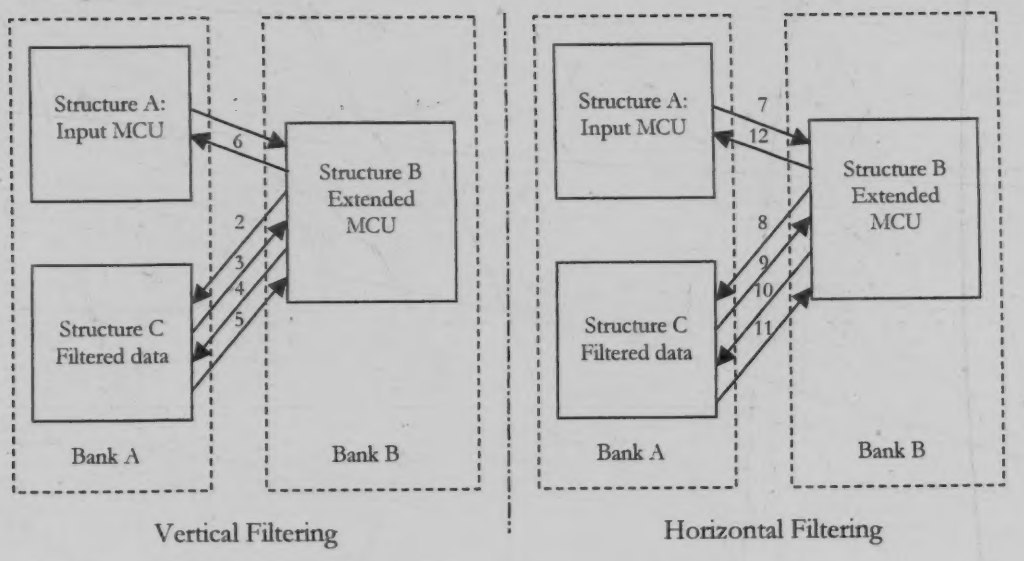

Figure 3.3-4 12 patterns of data flows in the DSP-RAM implementation of DWT

To make the figure easier to draw and understand, we further divide the 12 passes into two stages. The first stage is Vertical Filtering, which performs a 1-D DWT on each column of the 6 blocks in an MCU. The second stage is Horizontal Filtering, which performs 1-D DWT on each row of 6 blocks in MCU. In the algorithm, we defined a data structure A to store the source/destination MCU, a data structure B to store the extended arrays (which also constitute an extended MCU), and a data structure C to store filtered intermediate data. Because the source and destination of a vector operation of DSP-RAM can not be in the same memory bank due to the structure hazards of the PE's pipeline, we have to put structure B in a different memory bank from structures A and C.

In pass 1, we transfer all the columns from A to B and extend the columns as we introduced in **Figure 3.3-3**. In passes 2 and 4, we filter the extended arrays with high-pass and low-pass filters and store the results into C. In passes 3 and 5, we update the elements of the extended arrays in B with the filtered data in C. In pass 6, we update all the columns of the 6 blocks in A with truncated arrays in B. After pass 6 we have finished Vertical Filtering.

Horizontal Filtering is similar to Vertical Filtering except in pass 7 and pass 12. In pass 7 we transfer all of the rows (instead of the columns) from A to B. In pass 12, we not only store the truncated arrays back to A, we also de-interleave the coefficients so that the coefficients are reorganized into four blocks (LL, LH, HL and HH) and are ready for the processing at the next level.

3.3.3 Processing Flow of the DSP-RAM Implementation of DWT

In the 12 passes of data flows shown in **Figure 3.3-4**, 6 passes (pass 1, 3, 5, 7, 9, 11) involve data movement from bank A to bank B, two passes (pass 6, 12) involve data movements from bank B back to bank A, passes 2 and 8 involve high-pass filtering operations, and passes 4 and 10 involve low-pass filtering operations. In fact, there are only 4 distinct patterns of process flow in the DSP-RAM implementation of DWT. The high-pass and low-pass filtering operations are described in Equation 3.3-1, and they can be abstracted as shown in Equation 3.3-2 and Equation 3.3-3.

$$Q = A \times X + (B \times Y + C \times Z) >> 1 \qquad \text{Equation 3.3-2}$$

$$Q = A \times X + (B \times Y + C \times Z + D) >> 2 \qquad \text{Equation 3.3-3}$$

These two equations both have similar form as Equation 3.1-4, therefore their control algorithms are similar with what is shown in **Table 3.1-4**, except that Equation 3.3-2 takes three clock cycles to get the result while Equation 3.3-3 takes four.

The control state machine for our DWT implementation is shown in **Figure 3.3-5**. The numbers associated with arrows in the figure are the pass numbers that we introduced in **Figure 3.3-4**. The arrows define the direction of state transitions, and the numbers define the condition for each transition. For example, for the arrow pointing from state Start to state MOV0, the number 1 means the controller will transit from state Start to state MOV0 when we need to process pass 1 of the data flow.

We use four states (HPF0-3) to handle the high-pass filtering process defined by Equation 3.3-2. State HPF0 is a preparation state; the processing is done in the loop formed by the three states HPF1, HPF2 and HPF3. Once we finish the processes for pass 2 or 8, the control state will transit back to MOV0 to process pass 3 or 9.

Similar to high-pass filtering, we use five states (LPF0-4) to handle the low-pass filtering process defined by Equation 3.3-3. LPF0 is also a preparation state and the remaining four states constitute a processing loop.

At state MOV0, we perform data transfers from bank A to bank B for passes 1, 3, 5, 7, 9 and 11 of the data flows. At state MOV1, we perform data transfers from bank B to bank A for passes 6 and12. When we finish pass 12 at state MOV1, the control state will transit to state End and finish the whole DWT process.

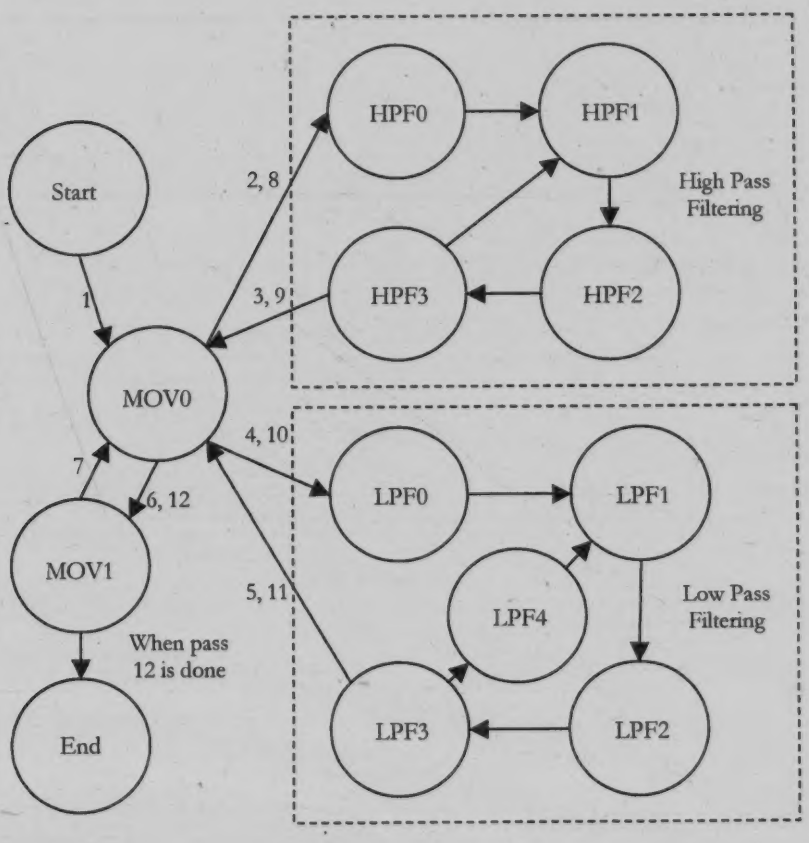

Figure 3.3-5 State transition diagram for the control of DWT on DSP-RAM

3.3.4 Clock Cycles for the DSP-RAM Implementation of DWT

By using the data flow and the processing flow of the DWT introduced in Section 3.3.2 and Section 3.3.3, we get a highly scalable algorithm for DWT which can be readily

embedded into our JPEG encoder implementation. The number of clock cycles needed for each of the 12 processing passes is listed in **Table 3.3-1**.

Table 3.3-1 Clock cycles needed for a PE to perform a DWT on a MCU

Pass	Process pattern	Clock cycles per operation	Number of operations	Clock cycles
1	Data movement	1	11×8×6	11×8×6+5=533
2	High-pass filtering	3	5×8×6	5×8×6×3+1+5=726
3	Data movement	1	5×8×6	5×8×6+5=245
4	Low-pass filtering	4	4×8×6	4×8×6×4+5=773
5	Data movement	1	4×8×6	4×8×6+5=197
6	Data movement	1	8×8×6	8×8×6+5=389
7	Data movement	1	11×8×6	11×8×6+5=533
8	High-pass filtering	3	5×8×6	5×8×6×3+1+5=726
9	Data movement	1	5×8×6	5×8×6+5=245
10	Low-pass filtering	4	4×8×6	4×8×6×4+5=773
11	Data movement	1	4×8×6	4×8×6+5=197
12	Data movement	1	8×8×6	8×8×6+5=389
Total:				**5726**

In this chapter, we analyzed the facts that affect the performance of a DSP-RAM implementation, and described how we implemented the JPEG encoder and the DWT function on a DSP-RAM system. Before we move on to the evaluation and analysis of our DSP-RAM implementations, we need to consider hardware aspects of the DSP-RAM architecture, such as how many PEs we could put into a chip with today's process technology, and how fast the chip could be made to operate. To determine this information, we synthesized a DSP-RAM architecture on a Virtex II pro FPGA. In next chapter, we will introduce our FPGA implementation of DSP-RAM in detail.

Chapter 4

FPGA IMPLEMENTATION OF THE DSP-RAM ARCHITECTURE

By providing a fast and cost-efficient path from concept to implementation, FPGA technology plays a very important role in VLSI architecture design and research. Especially in recent years, the major FPGA vendors have enhanced the basic programmable logic fabric with pre-designed cores, such as microprocessors, memory blocks and DSPs, into their FPGA architectures. Large gate capacities and high-performance cores reduce the time for building a system on a single chip and effectively improve the performance of key components in system. This makes the FPGA an ideal platform for prototyping DSP-RAM implementations.

In our research, we synthesized the DSP-RAM architecture on a Virtex II Pro FPGA. By completing an FPGA implementation of the DSP-RAM architecture, we utilized the latest process technology and design tools, conservatively evaluated the resource bottlenecks, and determined the maximum working frequency of the DSP-RAM. In this chapter, we will discuss our implementation in detail.

Section 4.1 introduces our design platform and the design target. Section 4.2 details the implementation of DSP-RAM, including the top-level architecture, key components, etc. Section 4.3 analyzes the resource requirements and performance of the FPGA implementation. Section 4.4 concludes this chapter.

4.1 Design Platform and Design Target

We used Xilinx Virtex-II Pro FPGAs as our hardware design platform. We also used Xilinx ISE 5.1 as our hardware development environment. The synthesis technology we used is XST (Xilinx Synthesis Technology). **Table 4.1-1** shows the on-chip resources that are available on current members of the Virtex-II Pro family of FPGAs [39].

Table 4.1-1 Resources list for the Virtex-II Pro Platform FPGAs

Feature/Product	XC 2VP2	XC 2VP4	XC 2VP7	XC 2VP20	XC 2VP30	XC 2VP40	XC 2VP50	XC 2VP70	XC 2VP100	XC 2VP125
Logic Cells	3,168	6,768	11,088	20,880	30,816	43,632	53,136	74,448	99,216	125,136
Block RAM (Kbits)	216	504	792	1,584	2,448	3,456	4,176	5,904	7,992	10,008
18x18 Multipliers	12	28	44	88	136	192	232	328	444	556
Digital Clock Management Blocks	4	4	4	8	8	8	8	8	12	12
Config (Mbits)	1.31	3.01	4.49	8.21	11.36	15.56	19.02	25.6	33.65	42.78
PowerPC Processors	0	1	1	2	2	2	2	2	2	4
Max Available Multi-Gigabit Transceivers	4	4	8	8	8	12*	16*	20	20*	24*
Max Available User I/O	204	348	396	564	644	804	852	996	1164	1200

In our implementation, we chose the XC2VP40-7 FPGA. We did not choose a larger chip because a larger chip would otherwise take much longer for synthesis and/or Place-And-Route (PAR). Since the DSP-RAM architecture is highly scalable and all the FPGAs in **Table 4.1-1** are compatible with each other, there would be no significant difference if we had chosen to implement DSP-RAM architecture on a different device in **Table 4.1-1** except the number of implementable PEs.

The main objective of our implementation exercise is to pursue the maximum performance based on the chosen platform.

4.2 Detailed Implementation of DSP-RAM on FPGA

4.2.1. Top-level Architecture

The top-level architecture of our FPGA implementation of DSP-RAM is shown in **Figure 4.2-1**. Compared to the top-level DSP-RAM architecture in **Figure 2.2-1**, there are two major modifications. First, our implementation specifies a tree-structured globe bus architecture instead of a strictly linear bus. A tree-structured bus can accommodate more PE and RAM blocks than a single linear bus architecture. Every branch of the bus tree can accommodate the same number of PE and RAM blocks. When the loads on a branch reach a performance limit, we can always add more branches to accommodate more PE and RAM blocks. Second, our implementation removes the shift register network from the original architecture. The shift register network provides

nearest-neighbor intercommunications among PEs organized in a one-dimensional array. As we discussed in **Section 3.1.2**, intercommunication among PEs is generally what we want to avoid because it tends to increase the overhead of a parallel algorithm. In our DSP-RAM implementation of JPEG and DWT, we successfully avoid intercommunication among PEs by dividing the incoming data into small units and processing each unit independently on each PE. Therefore, the shift register network is unnecessary for our target applications. Besides, when the PEs are organized into a tree structure, the shift register network increases the complexity of timing design and place and route due to the timing differences of different branches.

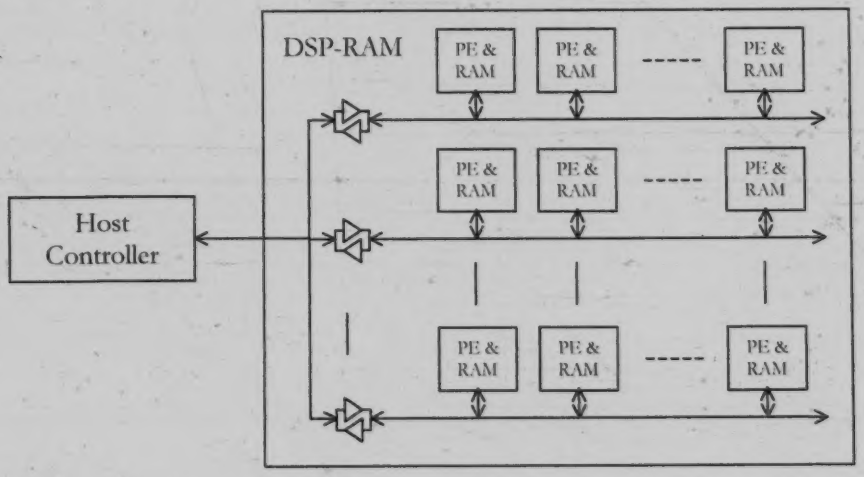

Figure 4.2-1 Top-level architecture of the FPGA implementation of DSP-RAM

The top-level architecture of a PE is shown in **Figure 4.2-2**. This figure only shows the key components and data path of PE; control signals have been omitted here for clarity.

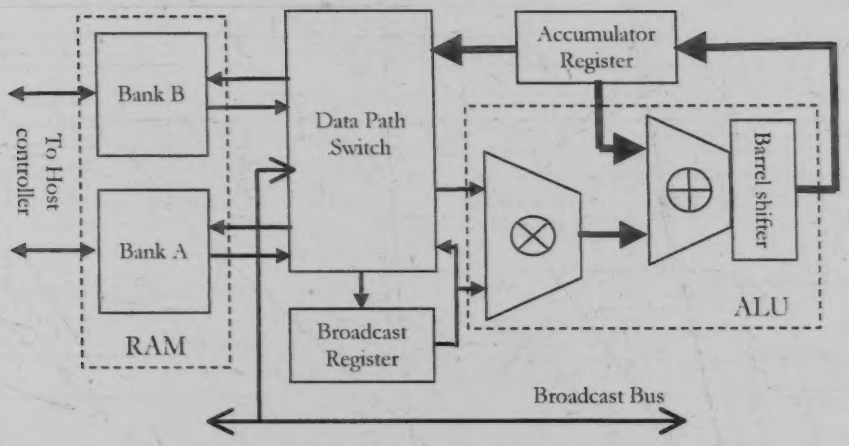

Figure 4.2-2 Top-level architecture of the FPGA implementation of a PE

Compared to the architecture in **Figure 2.2-2**, we have simplified the top-level architecture into three major components and two registers: RAM component, ALU component, data path switch component, broadcast register and accumulator register. The RAM component contains two banks of dual-port synchronous 1K×16 SRAM memory blocks. The ALU component contains a 16-bit×16-bit multiplier, a 48-bit accumulator and a 48-bit barrel shifter. The data path switch component is a combinational logic block that controls the interconnections among the other two components, two registers and the broadcast bus. An important modification to the original architecture in **Figure 2.2-2** is that, instead of using a bidirectional data bus for every memory bank at its memory interface at PE side, we used two separate sets of data buses for read and write operations. This modification, although it can not resolve the structure hazards of the DSP-RAM architecture, reduces the levels of logic at the memory interface, and greatly improves the maximum working frequency.

4.2.2. Implementation of the RAM Component

There are two kinds of memory in Vertex-II Pro FPGAs that can be used to implement DSP-RAM's memory blocks: Block SelectRAM (BRAM) and Distributed SelectRAM. Distributed SelectRAM provides "shallow" RAM throughout the chip. These RAMs are small in capacity but closer to the logic. BRAMs are dedicated memory blocks with a capacity of 18 Kb. Each block operates as a fully synchronous and true dual-port memory with a clock frequency of up to 300 MHz. BRAMs can be configured variously

as 16K×1, 8K×2, 4K×4, 2K×9, 1K×18 and 512×36 memory blocks. Also, these memory blocks can be combined together to create wider or deeper memories.

In our implementation, we used BRAMs to implement the memory blocks in DSP-RAM because they satisfied our capacity requirement while providing fast performance access. We configure each BRAM as a 1K×18 memory block and use them as a 1K×16 dual-port memory block in our implementation. Therefore two BRAM blocks can create one 2-Kbit RAM component for each PE. Since XC2VP40 has 192 BRAM blocks on chip, the number of RAM components that we can create on XC2VP40 is 96.

4.2.3. Implementation of the ALU Component

The ALU component contains a 16-bit×16-bit multiplier, a 48-bit accumulator and a 48-bit barrel shifter. Since the Virtex-II Pro platform has a large number of optimized 18-bit×18-bit multiplier blocks, we directly use them to implement a 16-bit×16-bit multiplier for the ALU component. These predesigned and preconnected multiplier blocks are optimized for high speed and consume less power than an 18-bit×18-bit multiplier implemented with configurable logic gates in the FPGA fabric.

Since there is no dedicated hardware support for the 48-bit accumulator and the 48-bit barrel shifter in Virtex-II Pro platform, we implemented them using configurable logic.

Xilinx provides a design tool called Core Generator, with which a developer can generate many function components by simply specifying values for the various component parameters. The netlists generated by the Core Generator specify the logic design and physical layout of these user-parameterized components and can be integrated into a large design with guaranteed performance.

In our implementation, we use Core Generator to implement the 32-bit multiplier and 48-bit accumulator. Because the current version of Core Generator does not provide support for barrel shifters, we wrote our own RTL codes in VHDL and synthesized designs into the chosen device.

4.2.4. Implementation of the Data Path Switch Component

Figure 4.2-3 shows the architecture of the data path switch component. This component uses four multiplexers to control the connections among the memory

blocks, broadcast bus, multiplier, broadcast bus register and accumulator register. These multiplexers are controlled directly by signals in the instruction bus, which is omitted in this figure.

Data path switch components are also implemented in programmable logic.

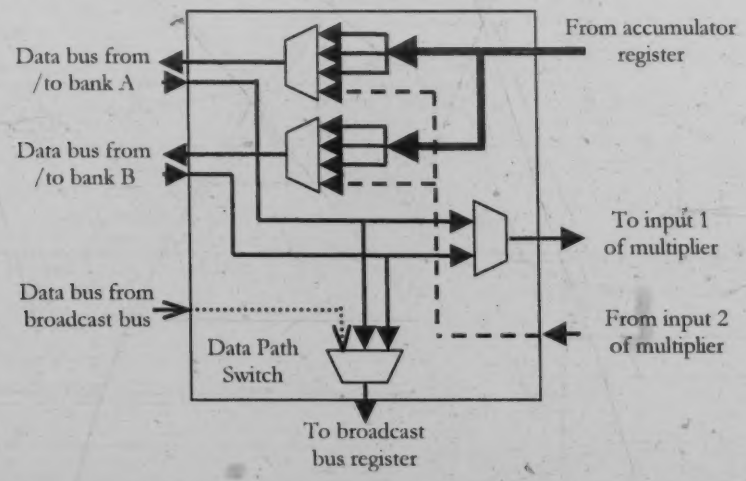
From accumulator register
Data bus from /to bank A
Data bus from /to bank B
To input 1 of multiplier
Data bus from broadcast bus
From input 2 of multiplier
Data Path Switch
To broadcast bus register

Figure 4.2-3 Architecture of the data path switch component

4.3 Analysis of Performance and Resource Requirements

In this section, the performance and resource analyses for individual components, individual PE, and arrays of PEs are presented based on the synthesis reports and post-PAR reports generated by the Xilinx ISE 5.1 software. We also analyze how DSP-RAM's size affects its performance and resource requirements.

Table 4.3-1 shows the performance and resources for key components in a PE.

Table 4.3-1 Performance and resource requirements for major components

Component	Maximum frequency	Critical resources
RAM	1.94 GHz (0.515 ns)	2 BRAM, 2 LUT, 108 FF, 64 slices
MAC	247.097 MHz (4.047 ns)	1 MULT, 96 LUT, 112 FF, 89 slices
Barrel Shifter	291.630 MHz (3.429 ns)	447 LUT, 192 FF, 239 slices
Data Path Switch	7.369 ns (0.797 ns)	152 LUT, 85 slices

1. BRAM: Block SelectRAM, LUT: Look Up Table, FF: Flip Flop, MULT: Multiplier Block, slice: Configurable logic.
2. The Data Path Switch component is implemented in combinational logic, 7.369 ns is the delay of its critical path including the I/O delays.

3. The 7.369 ns delay in the Data Path Switch component includes the delay of the I/O buffers (6.572 ns), which should not be counted when the Data Path Switch component is integrated inside of a system. Therefore, a more useful estimation of its critical path should be 0.797 ns.

These data were obtained by synthesizing each component individually on the Xilinx 2vp40ff1152-7 FPGA. The estimates for resource requirements are usually accurate, but the estimates for the timing of designs may not be accurate because they do not take the effects of PAR (Place And Route) into account. Hence, although the data shows that the slowest component is the MAC component, PAR may change the relative performance of these components when they are merged together into a system.

Table 4.3-2 Performance and resources for a single PE and groups of PEs

Component	Maximum frequency	Critical resources		
		MULT	BRAM	Slices
One PE	279.642 MHz	1/192 (1%)	2/192 (1%)	424/19392 (2%)
16 PEs	227.428 MHz	16/192 (8%)	32/192 (16%)	7122/19392 (36%)
2×16 PEs	226.449 MHz	32/192 (16%)	64/192 (33%)	14263/19392 (73%)

Table 4.3-2 shows the performance and resource requirements for the implementations of a single PE, an array of 16 PEs, and a 2×16 matrix of 32 PEs. In all cases we assume the 2vp40ff1152-7 FPGA. These data are collected from the post-PAR reports; therefore, the effects of PAR are taken into account in these numbers. It's evident that the slices, which are the programmable logic elements on the chip, are the critical resources for our design. Although we still have large amounts of on-chip memory and many multiplier blocks available, we could not utilize them efficiently because the proportion of available on-chip resources does not closely match the requirements of DSP-RAM. This problem would tend to increase the cost of a DSP-RAM chip, but the inefficiency in hardware utilization could be overcome once the DSP-RAM implementation is transferred to a custom ASIC.

When the utilization of on-chip resources increases, the difficulty for optimizing the PAR process is also increases. The performance of the resulting FPGA implementation usually degrades when more PEs are integrated. However, this problem could also be overcome by implementing DSP-RAM on an ASIC.

When we integrate all components of a single PE and iteratively reduce the timing constraints for the clock period down to 3.5 ns, then the place and route tool of ISE 1.5i returns an implementation with a minimum clock period of 3.576 ns (279.642 MHz), consuming no more than 2% of the on-chip logic resources. The critical path of the design is in a pipeline stage of the multiplier component, corresponding to the most significant bit of the data path (32 bits). All of the delay components of the critical path are in one logic slice and all the timing delays are close to the minimum switching characteristics posted in the data sheets of the Virtex-II Pro; hence the PE implementation is close to being fully optimized for the Virtex-II Pro platform.

When we integrate 16 PEs into a PE array, and then set the timing constraints of the target clock period to 3.6 ns, we get an optimized implementation with a minimum clock period of 4.397ns (227.428 MHz). It consumes 36% of the logic slices, 16% of the BRAMs and 8% of the block multipliers. Since the utilization of the logic slices increases when we put more PEs into one chip, the speed of the circuit degrades because the resource limits begin to interfere with the optimization of the place and route process.

When we further increase the number of PEs on a chip, by integrating two 16-PE arrays into a PE matrix, the performance of circuits further degrades while the utilization of slices increases to 73%. However, by carefully setting the clock period time constraint to 4.45 ns, we eventually get a relatively well optimized implementation with a minimum clock period of 4.416 ns (226.449 MHz).

Table 4.3-3 Timing closure of an FPGA implementation for a 2×16 PE matrix

Constraint (ns)	Timing closure (ns)	Critical path
4.3	4.925	Inside of barrel shifter
4.41	5.198	RAM to broadcast register
4.42	4.506	RAM to multiplier
4.43	4.735	Inside of barrel shifter
4.44	4.573	Multiplier to accumulator
4.45	***4.416 (Constraint met)***	***RAM to broadcast register***
4.46	4.449 (Constraint met)	Multiplier to accumulator
4.5	4.469 (Constraint met)	RAM to broadcast register

Table 4.3-3 shows the process of using timing constraints to optimize the place and route of the FPGA implementation of the 2×16 PE matrix. If we tighten the timing constraint from 4.45 ns to 4.44 ns or 4.43 ns, or loosen the timing constraint from 4.45 ns to 4.46 ns or 4.5 ns, the performance of the circuit will degrade to some extent, while the critical path of the resulting circuit will shift around to different parts of the circuit. The change of the critical path for different timing constraints indicates that there is no path whose delay is significantly larger than other paths. This likely implies that the design is well optimized.

4.4 Conclusions

In this chapter, we introduced an optimized FPGA implementation of DSP-RAM. By implementing DSP-RAM on a Virtex-II Pro FPGA, we utilized the latest process technology and the latest FPGA development platform. By modifying the top-level architecture of DSP-RAM into a tree-structured architecture, we reduced the loads of every global bus line, and increased the capacity of DSP-RAM architecture. By implementing memory blocks with dual-port memory, we reduced the complexity of the communication path between DSP-RAM and host controller. By separating the read and write memory ports in each PE, we further increased the memory bandwidth, reduced the level of logic in the original critical path, and also reduced the resource requirements for on-chip logic.

The resulting implementation is a balanced, optimized DSP-RAM implementation for the Virtex-II pro FPGA platform. We improved the maximum operating clock frequency from 50 MHz to 200 MHz and implemented 32 PEs and 64 memory blocks on one 2vp40 FPGA device.

In the future, DSP-RAM could be more efficiently implemented on an ASIC for the sake of performance and cost. More functions, such as the pruned left-right shift register network, may need to be added back into the architecture. With improvements in FPGA technology and the growing maturity of the development tools, it should be possible to implement the host controller and DSP-RAM on one single chip.

In the next chapter, we will evaluate and analyze DSP-RAM's ability to perform JPEG encoding based the hardware information we obtained in this chapter.

Chapter 5

EVALUATION

In Chapter 3, we discussed the factors that affect the performance of a DSP-RAM implementation of the JPEG encoder and the DWT algorithm. For some of the function blocks, we compared our DSP-RAM implementations with potential implementations on a load-store superscalar architecture. However, because there are many uncertainties, we can only get a rough idea of the advantages and disadvantages of our implementations.

To fully understand DSP-RAM's efficiency, we need to evaluate nontrivial system-level DSP-RAM implementations. In this chapter, we will evaluate the performance of our parallel JPEG encoder and the DWT algorithm, implemented on the DSP-RAM architecture, against their sequential counterparts implemented on a superscalar microprocessor in a typical PC architecture.

Section 5.1 introduces the hardware and software platforms for which we will perform our evaluations. We also describe the methodologies that we applied in our evaluation and analysis. Section 5.2 gives the detailed evaluation results. Section 5.3 summarizes the analysis of the results.

5.1 Evaluation Platform and Methods

5.1.1. Evaluation Platform and Methods for DSP-RAM Implementations

Our DSP-RAM implementation was evaluated on a DSP-RAM simulator. The DSP-RAM simulator is a C++ static library, which contains object classes that specify the internal structure and component behaviors of sub-parts of DSP-RAM. When we simulate applications on DSP-RAM, we generate an object instance of the DSP-RAM class and then call member functions of the object to simulate hardware operations. During the simulation, the DSP-RAM object records the number of clock cycles that have been run so far so that the overall number of clock cycles for the application will be available at the end of simulation. Since the DSP-RAM simulator simulates the exact operations that would happen on DSP-RAM hardware at each clock cycle, the number

of clock cycles recorded in the simulation will be exactly the same as the number of clock cycles needed by the actual hardware.

Therefore, we can calculate the runtime of a DSP-RAM implementation with Equation 5.1-1, where we can use the maximum operating clock frequency of our FPGA implementation for DSP-RAM to calculate the clock cycle time.

$$Run\_time = clock\_cycles \times clock\_cycle\_time \qquad \text{Equation 5.1-1}$$

In the simulation, we assume that the host controller runs at the same clock speed as the DSP-RAM, issues instructions at each clock cycle and does nothing else when it is controlling DSP-RAM. But in a real implementation, it is possible to issue the instructions through some extra hardware. The host controller only needs to trigger a hardware sequencer to start issuing instructions. The host could then process some tasks in parallel with DSP-RAM. The DSP-RAM may inform the host controller with an interrupt signal when the DSP-RAM has finished the task assigned on it. Since we didn't consider this potential improvement, the runtime data obtained in our simulation would be conservative (assuming no changes in the clock period).

5.1.2. Evaluation Platform and Methods for Sequential Implementations

In our research, we chose the JPEG encoder from IJG (Independent JPEG Group) as the benchmark application for Intel's Pentium 4 microprocessor (the older version without support for Hyper-Threading Technology). IJG is an informal group that writes and distributes a non-commercial, widely used, actively maintained C library for JPEG image compression. Its codec is well known because of its portable, effective, stable and reliable codes.

The hardware and software platform on which we ran the convention IJG JPEG encoder and DWT algorithm is shown as **Table 5.1-1**.

Table 5.1-1 Hardware and software platform for evaluating sequential algorithms

Name	Description
Processor	Intel Pentium 4 Willamette (0.18-μm) processor
Clock Frequency	1.596 MHz
L1 data cache	8 KB, 4-way, 32-byte lines
L1 code cache	12 KB-μOps, 4-way, 32-byte lines
L2 cache	256 KB, 8-way, 64-byte lines
Front system bus	100 MHz
Main memory	Rambus, 512 MB
SIMD extension	SSE2
BIOS	07/18/01
Operating system	Red Hat Linux 7.1 2.96-98
Compiler	gcc version 2.96 20000731

When we compile source codes with gcc v2.96, we enabled switch –O3 to obtain the highest optimization level. We also enabled switch –pg so that we can use the Gnu Profiler to measure the runtime of programs. To increase the accuracy of the measurement, source codes were modified so that they processed an image 100,000 times per run. The runtime results generated by the GNU profiler were divided by 100,000 to get averaged run-time statistics. The measurements were taken for images of size 227×149, 800×600, 1024×768 and 1280×1024, repeated 20 times for each image. The smallest overall runtime was picked for each image because the overheads from GNU profiler and other processes are the smallest at this situation.

Although the IJG codec does not utilize the Streaming SIMD extensions available on the Pentium 4, Intel has studied the speed-up that their extensions could provide for the IJG codec. Therefore, we can get the runtime statistics of the enhanced IJG codec by multiplying the runtime of original IJG encoder with the speed-up factors published by Intel. **Table 5.1-2** gives the research data released on Intel's website.

The channel number stands for the number of samples that an image used to represent one pixel. For example, if an image uses RGB values to represent its pixels, the image has three channels. If the image uses an additional channel (alpha channel) to represent the transparency of pixels, then the image's channel number increases to 4.

The sampling number stands for the subsampling ratio for the three channels. A sampling number of 444 means no subsampling is required for all three channels. The downsampling process described in Section 2.1.2 and Section 3.2.4 has a sampling ratio of 411.

IPP stands for Integrated Performance Primitives, which is a low-level basic software layer that hides the difference of diverse hardware platforms and enables developers to utilize the advanced features of hardware platforms without the need to program at the assembly level [40]. The performance numbers in the columns of IJG or IJG+IPP give the CPU cycles per pixel. Hence the numbers are independent of the operating frequency of the target microprocessor.

As shown in **Table 5.1-2** [40], for a 3-channel JPEG compression with a subsampling rate of 411, Intel Pentium 4's Streaming SIMD extension can speed up the IJG encoder by a factor of 2.7.

Table 5.1-2 Performance numbers obtained for the original version of the IJG library and for the modified version with the Intel Integrated Performance Primitives (IPP)

JPEG parameters			Pentium III processor 500 MHz			Pentium 4 processor 1400 MHz		
Codec	Channels	Sampling	IJG	IJG+IPP	Speedup	IJG	IJG+IPP	Speedup
Encoder	1	444	200.3	81.6	2.4	270.8	82.7	3.2
	3	444	554.4	230.3	2.4	742.1	248.8	2.9
	3	422	385.2	167.4	2.3	504.5	181.3	2.7
	3	***411***	***300.4***	***137.5***	***2.1***	***391.8***	***143.5***	***2.7***
	4	444	810.2	593.4	1.3	1056.5	341.3	3.0
	4	422	620.2	254.0	2.4	803.1	267.0	3.0
	4	411	524.3	274.1	1.9	678.8	230.7	2.9
Decoder	1	444	81.3	51.8	1.5	110.4	60.3	1.8
	3	444	232.1	140.6	1.6	307.2	161.0	1.9
	3	422	171.2	104.3	1.6	225.8	120.6	1.8
	3	411	141.8	89.8	1.5	180.4	103.4	1.7
	4	444	325.3	194.1	1.6	423.1	219.6	1.9
	4	422	253.5	158.1	1.6	332.6	176.8	1.8
	4	411	219.7	141.4	1.5	289	162.9	1.7

5.2 Evaluation Results

5.2.1. Run-time Statistics for DSP-RAM Implementations

To clarify the relationship among the number of PEs, image size and performance of our DSP-RAM JPEG encoder, we simulated modified DSP-RAM architectures with different numbers of PEs to process images of different sizes.

The runtime statistics for parallel parts of the DSP-RAM JPEG encoder are shown in **Table 5.2-1**. These data are simulated with the methods introduced in section 5.1.1. Clearly increasing the number of PEs improves the performance of the parallel parts of the DSP-RAM JPEG encoder. The same results are plotted graphically in **Figure 5.2-1**.

Table 5.2-1 Runtime statistics for the parallel parts of the DSP-RAM JPEG encoder

Number of PE	Clock cycles/Runtime @200 MHz (ms)							
	227×149 (150 MCU)		800×600 (1900 MCU)		1024×768 (3072 MCU)		1280×1024 (5120 MCU)	
1	1,963,350	9.82	24,869,100	124.35	40,209,408	201.05	67,015,680	335.08
2	981,675	4.91	12,434,550	62.17	20,104,704	100.52	33,507,840	167.54
4	497,382	2.49	6,217,275	31.09	10,052,352	50.26	16,753,920	83.77
8	248,691	1.24	3,115,182	15.58	5,026,176	25.13	8,376,960	41.88
16	130,890	0.65	1,557,591	7.79	2,513,088	12.57	4,188,480	20.94
32	65,445	0.33	785,340	3.93	1,256,544	6.28	2,094,240	10.47
64	39,267	0.20	392,670	1.96	628,272	3.14	1,047,120	5.24
80	26,178	0.13	314,136	1.57	510,471	2.55	837,696	4.19
96	26,178	0.13	261,780	1.31	418,848	2.09	706,806	3.53
112	26,178	0.13	222,513	1.11	366,492	1.83	602,094	3.01
128	26,178	0.13	196,335	0.98	314,136	1.57	523,560	2.62

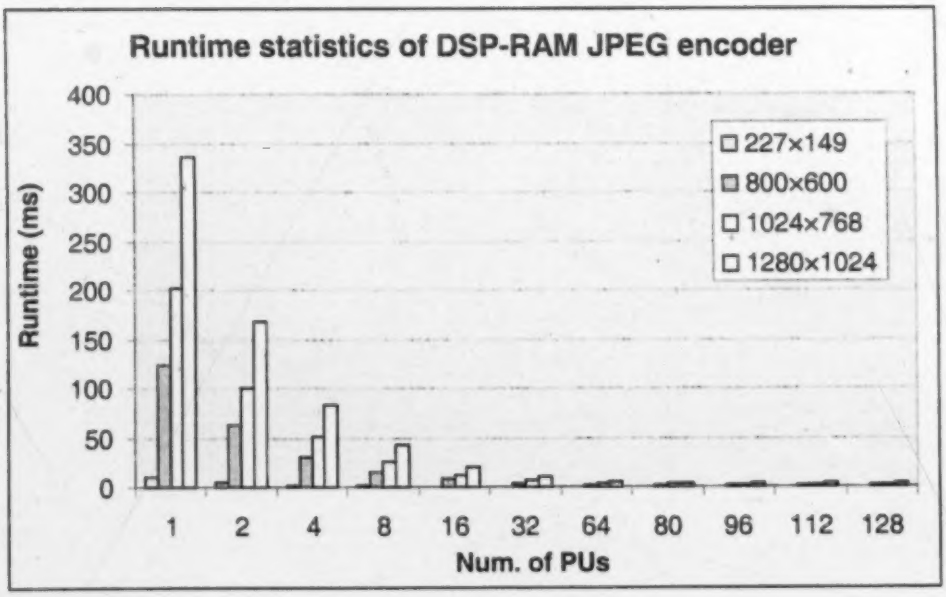

Figure 5.2-1 How the number of PEs affects the performance of the DSP-RAM JPEG encoder

Since we use the MCU as the least data unit for image compression, and since we assign each MCU to a PE for processing, the runtime of our algorithm is directly related to two numbers: the number of MCUs in the image and the number of PEs in the DSP-RAM architecture.

For a chosen image, the MCU number is fixed. Doubling the number of PEs will generally halve the runtime of the DSP-RAM JPEG encoder. For example, for the image of size 1280×1024 in **Table 5.2-1**, which has 1280 MCUs, increasing the number of PEs from 1 to 128 will reduce the runtime of DSP-RAM JPEG encoder by a factor of about 128 times, from 335.08 ms down to 2.62 ms.

However, sometimes when the MCU numbers are small and each PE has only a few MCUs to process, doubling the number of PEs might not be able to reduce the load of each PE, hence the runtime might not be halved. For example, for the image of size 227×149 in **Table 5.2-1**, which has 150 MCUs, if we double the number of PEs from 32 to 64, the workload per PE will only be reduced from 5 MCUs to 3 MCUs, hence the runtime of the DSP-RAM JPEG encoder will only be reduced from 0.33 ms to 0.20 ms (3/5). For the same reason, when we increase the number of PEs from 64 to 80, the

workload per PE will be reduced from 3 MCU to 2 MCU and the runtime will be reduced from 0.20 ms to 0.13 ms. Finally, when we increase the number of PEs from 80 to 112, even to 149, each PE still have to process two rounds because we do not have enough PEs to process 150 MCUs in one round. Therefore increasing the number of PEs does not help to reduce the runtime in such a case.

Table 5.2-2 shows the distribution of runtimes for the parallel functions in the DSP-RAM JPEG encoder. Because the algorithms adopted to implement these functions on DSP-RAM are independent of the size and content of the image, the numbers in the table are also independent of them. Because of the limitations of the PE's ALU, we did not implement entropy encoding on DSP-RAM. Instead, we performed entropy encoding and file I/O operations on the host controller. Hence the speed of the sequential parts of the encoder depends on the performance of the chosen host controller.

Table 5.2-2 Runtime distribution for parallel functions in the DSP-RAM JPEG encoder

Color conversion	Subsampling	DCT transform	Quantization	Entropy encoding
27.41%	10.82%	47.03%	14.74%	Not parallel

5.2.2. Run-time Statistics for Sequential Implementations

For the compression of images of size 227×149, 800×600, 1024×768 and 1280×1024, the runtime statistics of the IJG JPEG encoder on the Pentium 4 host are shown in **Table 5.2-3**. These data are measured with the methods introduced in section 5.1.2.

Table 5.2-3 Runtime statistics for the IJG JPEG encoder

Function blocks	Runtime (ms)			
	227×149	800×600	1024×768	1280×1024
Color conversion	0.78 (16.32%)	9.92 (15.90%)	17.13 (13.06%)	27.73 (18.86%)
Subsampling	0.13 (2.80%)	1.58 (2.53%)	2.62 (2.00%)	4.35 (2.96%)
Forward DCT	1.31 (27.56%)	18.69 (29.95%)	30.89 (23.55%)	50.87 (34.59%)
Quantization	1.13 (23.76%)	14.75 (23.64%)	34.85 (26.57%)	31.52 (21.43%)
Entropy encoding	1.16 (24.32%)	15.00 (24.04%)	42.66 (32.52%)	28.44 (19.34%)
5 main functions	4.52 (94.76%)	59.94 (96.07%)	128.15 (97.69%)	142.91 (97.18%)
Others	0.25 (5.24%)	2.45 (3.93%)	3.03 (2.31%)	4.14 (2.82%)
Overall runtime	**4.77**	**62.39**	**131.18**	**147.05**
Compression ratio	**17.9**	**22.8**	**10.9**	**46.6**

When the resolution of the image increases, the image compression time for a 1.6-GHz P4 machine also increases. The five main functions take around 96% of the time of the whole process. The runtime distribution of the five main functions may vary for different images. For example, the image of size 1024×768 in **Table 5.2-3** is a picture with many sudden color changes. In other words, it has relatively large high frequency content. Because these high frequency components can not be simply substituted with zero in the quantization stage, the IJG encoder has to perform more divisions in the quantization stage and more bit-shifting operations in the entropy encoding stage; hence the proportion of runtime for these two stages is much higher than other cases, and the final compression ratio is much lower than other cases as well. In contrast, the image of size 1280×1024 in **Table 5.2-3** is a picture in which the color changes are mainly gradual and smooth. Most of its high frequency components are thus very small so that IJG encoder can substitute them directly with zero; hence the proportion of runtime for quantization and entropy encoding is smaller, and the compression ratio is greater.

5.3 Analysis of Results

5.3.1. Analysis for Parallel Parts of Encoding

In the DSP-RAM implementation of the JPEG encoder, we process color conversion, subsampling, DCT transform and quantization on DSP-RAM in parallel, and process

entropy encoding on host controller sequentially. Evaluation results for a comparison of the parallel parts of encoding on DSP-RAM versus its counterparts in the sequential IJG JPEG encoder are given in **Table 5.3-1**.

Table 5.3-1 Performance improvement for the parallel parts of encoding

Platform	Run times for the parallel parts of encoding (ms)			
	227×149	800×600	1024×768	1280×1024
P4 @ 1.6 GHz	3.35	44.94	85.49	114.47
DSP-RAM (32 PE @ 200 MHz)	0.33 (10.15×)	3.93 (11.44×)	6.28 (13.61×)	10.47 (10.93×)
DSP-RAM (128 PE @ 200 MHz)	0.13 (25.77×)	0.98 (45.86×)	1.57 (54.45×)	2.62 (43.69×)

For a DSP-RAM chip with 32 PEs working at a 200 MHz clock frequency, DSP-RAM generally gains a speedup of 10-13 times for the first 4 main functions of the JPEG encoding compared to a P4 system working at 1.6 GHz. If the DSP-RAM chip has 128 PEs, the speedup may be raised to around 50 times for large images. As we mentioned before, for small images, such as images with resolution of 227×149, where the number of MCUs is comparable to the number of PEs, the relationship between the speed-up and the number of PEs is more like a step curve, i.e. the performance may not be improved until the number of PEs reaches certain threshold. For an image of size 227×149, which has 150 MCUs, the runtime can be reduced to 0.13 ms when the number of PEs is increased to 75, and it will not be reduced further until the number of PEs reaches 150.

5.3.2. Analysis for Overall JPEG Encoding Process

Since parts of the encoding process are performed on the host controller of DSP-RAM, the overall performance of a DSP-RAM JPEG encoder is also affected by the computational power of the host controller. Therefore, choosing different host controllers for a DSP-RAM system may lead to different evaluation results. Before we start to evaluate the performance at the system-level, we have to reduce this uncertainty.

We see DSP-RAM as having two major types of applications in the future. First, DSP-RAM may be used as a co-processor that increases the computational power of the system. Second, DSP-RAM may be used together with a less powerful microprocessor to accelerate certain applications while reducing system cost and/or power consumption.

For the first types of applications, we could directly use the system microprocessor as the host controller. In the case of our evaluation, we chose the P4 as our host controller, and then determined how much speed-up a DSP-RAM co-processor could bring to the system. For the second types of applications, we could evaluate how much computational power the host controller would need to have to provide comparable computational power to a conventional P4 system.

Table 5.3-2 gives the estimates of the speed-up that a DSP-RAM co-processor could bring to a P4 system when performing JPEG encoding.

Table 5.3-2 Speed-up for a JPEG encoder on a Pentium 4 system enhanced by a DSP-RAM co-processor

Platform	Runtime (ms)			
	227×149	800×600	1024×768	1280×1024
P4[1] alone	4.77	62.39	131.18	147.05
P4[1] with DSP-RAM[2] (32 PEs)				
Parallel portion	0.33 (18.97%)	3.93 (18.38%)	6.28 (12.08%)	10.47 (24.32%)
Entropy encoding	1.16 (66.67%)	15.00 (70.16%)	42.66 (82.09%)	28.44 (66.06%)
others	0.25 (14.37%)	2.45 (11.46%)	3.03 (5.83%)	4.14 (9.62%)
Overall	**1.74 (2.74×)**	**21.38 (2.92×)**	**51.97 (2.52×)**	**43.05 (3.42×)**
P4[1] with DSP-RAM[2] (128 PEs)				
Parallel portion	0. 13 (8.50%)	0.98 (5.32%)	1.57 (3.32%)	2.62 (7.44%)
Entropy encoding	1.16 (75.82%)	15.00 (81.39%)	42.66 (90.27%)	28.44 (80.80%)
others	0.25 (16.34%)	2.45 (13.29%)	3.03 (6.41%)	4.14 (11.76%)
Overall	**1.53 (3.12×)**	**18.43 (3.39×)**	**47.26 (2.78×)**	**35.2 (4.18×)**

1. Pentium 4 @ 1.6 GHz.

2. DSP-RAM @ 200 MHz.

The runtime data for a Pentium 4 system was obtained from **Table 5.2-3**. The runtime data for Pentium 4 system enhanced with DSP-RAM was calculated by adding the runtime for processing entropy encoding on the P4 system, which we get also from **Table 5.2-3**, with the runtime for processing the other 4 main functions on DSP-RAM, which is from **Table 5.3-1**.

From **Table 5.3-2**, we observe the following factors:

1. The speed up provided by 32 PEs is around 2.5-3.4 times, depending on the characteristics of the image.
2. Compared with the data in **Table 5.1-2**, in which the SSE2 extension accelerated the Pentium 4 processor on JPEG processing around 1.7-3.2 times, a DSP-RAM (32 PEs) running at 200 MHz shows a little performance advantage over the SSE2 extension by providing a speed-up of 2.5-3.4 times.
3. For a video stream of resolution 800×600, it's possible for a DSP-RAM enhanced P4 system to achieve a performance of about 46.8 fps (frames per second) by only using JPEG compression technology.

Table 5.3-3 shows the performance requirement of the host controller for a DSP-RAM system to provide the same performance as a 1.6-GHz Pentium 4 system on JPEG encoding.

Table 5.3-3 Performance requirement for the host controller when a DSP-RAM competes with a Pentium 4

Platform	Runtime (ms)			
	227×149	800×600	1024×768	1280×1024
P4[1]	4.77	62.39	131.18	147.05
Entropy encoding	1.16 (24.32%)	15.00 (24.04%)	42.66 (32.52%)	28.44 (19.34%)
Others	0.25 (5.24%)	2.45 (3.93%)	3.03 (2.31%)	4.14 (2.82%)
Sum	1.41	17.45	45.69	32.58
DSP-RAM[2] (32 PEs)	4.77	62.39	131.18	147.05
Parallel portion	0.33	3.93	6.28	10.47
Sequential portion	**4.44 (3.15×[3])**	**58.46 (3.35×[3])**	**124.90 (2.73×[3])**	**136.58 (4.19×[3])**
DSP-RAM[2] (128 PEs)	4.77	62.39	131.18	147.05
Parallel portion	0.13	0.98	1.57	2.62
Sequential portion	**4.64 (3.29×[3])**	**61.41 (3.52×[3])**	**129.61 (2.84×[3])**	**144.43 (4.43×[3])**

1. Pentium 4 @ 1.6 GHz.

2. DSP-RAM @ 200 MHz.

3. Allowed slow-down.

If a DSP-RAM system is required to perform one JPEG encoding in the same time that a Pentium 4 machine at 1.6 GHz requires, then the host controller could be around 3.5 times slower than a Pentium 4 at 1.6 GHz. The 3.5 times performance margin may lead

to a significant reduction on system cost and/or power consumption. It also frees up the host to allow it to provide other services.

5.3.3. Approaches to Further Improve the DSP-RAM JPEG encoder

✧ Versatility or Computational Power

Conforming with Amdahl's Law, which repeated below as Equation -5.3-1, by performing around 70% of the processing of a JPEG encoder on a parallel 32-PE DSP-RAM co-processor, we speed up the IJG JPEG encoder by a factor of about 3 times.

$$Speedup_{overall} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \qquad \text{Equation 5.3-1}$$

In Equation 5.3-1, $Fraction_{enhanced}$ denotes the proportion of the calculation that is parallelizable, which can be found in **Table 5.3-2**. $Speedup_{enhanced}$ denotes the speed-up of the parallel part of the process against its sequential counterparts, which can be found in **Table 5.3-1**.

As shown in **Table 5.3-2**, the runtime for the parallel part of the process can be much less than 25% of the overall runtime. To further improve the performance of DSP-RAM JPEG encoder, we can either further improve the 25% of parallel processing by increasing the clock frequency or increasing the number of PEs, or we can increase the proportion of the parallel parts in the process by improving the versatility of DSP-RAM's PE.

Our evaluation suggests that the second way is more efficient. As we can see from Equation 5.3-1, the highest speed up we can achieve through the first way is the reciprocal of the proportion of the sequential parts, because the second term in the denominator of Equation 5.3-1 will tend to zero when $Speedup_{enhanced}$ increases. In our case, if we do not improve the versatility of DSP-RAM's PE, we will only be able to improve around 25% of the process. The highest speed-up achievable in this way will be less than 1.3 times. As shown in **Table 5.3-2**, when we increase the number of PE

by a factor of 4 times, the speed up only increases from around 2.5-3.4 times to around 2.8-4.2 times.

If we adopt the second way, say, increasing the portion of parallel parts from current 25% to 75% by performing part of the entropy encoding on DSP-RAM, i.e. $Fraction_{enhanced} = 50\%$ and keep the $Speedup_{enhanced}$ at around 10 times, we will be able to further improve the performance of the DSP-RAM JPEG encoder by a factor of 1.8 times. Therefore, the parallel JPEG encoder would have a speed-up of about 5.4 compared to perform the IJG JPEG encoder on a 1.6-GHz Pentium 4 system. Such a DSP-RAM system would have the computational power to compress video signals of resolution 1280×1024 with JPEG technology in real time (36.7 fps).

Hence, instead of spending transistor budgets on increasing the number of PEs and memory pairs to achieve the maximum speed up of 1.3 times, we would suggest investing transistors to increase the complexity and versatility of every PE.

✧ Problems with the Parallelization of the Entropy Encoding

As we discussed in section 2.1.2, entropy encoding involves three types of coding technologies: differential coding, run-length coding and Huffman coding. The problems of parallelizing entropy encoding on the DSP-RAM architecture mainly exist in the parallelization of the run-length coding and the Huffman coding stages.

First, some useful operations are not supported by the current DSP-RAM architecture. For example, the Huffman coding requires performing table look-up operations which have not been supported. As another example, the run-length coding needs a counter to count how many consecutive zeros have been read from the memory. This counter should be increased or cleared or read according to two consecutive values read from the memory. With only one comparator and no place to store the comparison result, it's impossible to perform such an operation on two consecutive values.

Second, these two coding schemes require performing different operations on different PEs depending on the processed data. In the current DSP-RAM architecture, every PE receives the same instruction and performs the same operations. Sometimes, a PE can discriminate itself from others by disabling the memory write when others are writing. But this method is not suitable here. For example, when processing the run-length

coding in parallel on different PEs, a PE may encounter a zero-valued AC coefficient and may need to increment its counter while another PE may encounter a non-zero AC coefficient and then needs to record the counter value into memory and clear the counter value in register. Here we not only need to control which PE should write to its memory, but also need to control the counters, some of them may increase, some of them may be cleared.

To solve these problems, extra hardware has to be added into DSP-RAM architecture.

✧ Parallelization of the Differential Coding

After the quantization stage, every PE has six 8×8 blocks of coefficients stored in its local memory. The coefficients at the position (1, 1) in the 8×8 blocks are DC coefficients. The differential coding will subtract each DC coefficient of the previous block from the DC coefficient of the current block and store the difference at position (1, 1) after coding. **Figure 5.3-1** shows the data flow of a possible DSP-RAM implementation of differential coding. The small grey blocks indicate the position of the DC coefficients. The arrows indicate the direction of data transferring.

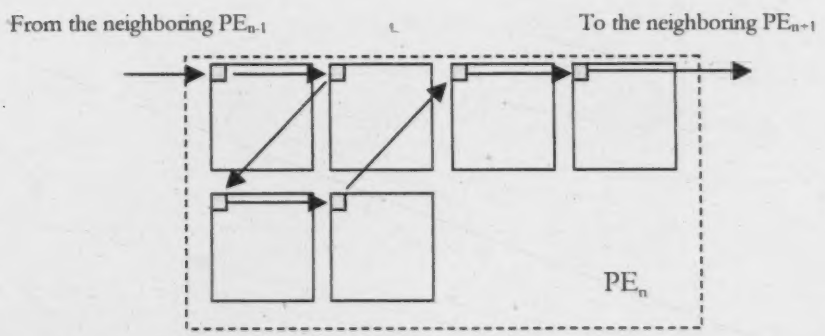

Figure 5.3-1 The DSP-RAM implementation of the differential coding

✧ Parallelization of the Run-Length Coding

In the entropy coding stage of JPEG, run-length coding is used to compress the zero-valued AC coefficients. The coder needs to scan the 8×8 block in a zigzag order and find the number of the consecutive zero values. Instead of encoding those zero

values into the final bit stream, JPEG encodes and embeds the number of the consecutive zero values.

To parallelize run-length coding on the DSP-RAM architecture, a PE should be able to detect the beginning and the end of a string of zeros and be able to start and stop counting based on the result of detection.

A possible approach to realize zero-series detection is to modify the branch control circuit of the PE as shown in **Figure 5.3-2**.

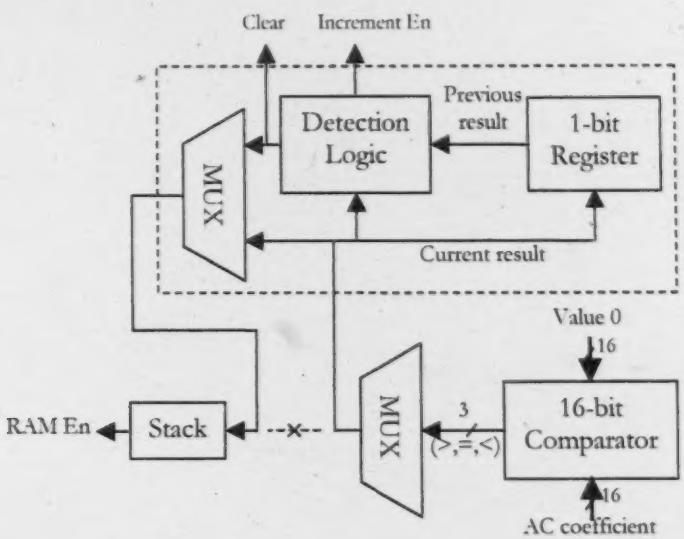

Figure 5.3-2 Modified branch control circuit for run-length coding

The truth table for the detection logic is shown in **Table 5.3-4**. When the current/previous AC coefficient is zero, the current/previous result will be 1. The Increment En signal enables the Accumulator. The Clear signal indicates that the end of the zero sequence is detected. It is used to control the RAM En so that the number of the zeros in the zero serial, which is stored in the Accumulator Register, can be written into the memory. It is also used to clear the value in the Accumulator Register to zero.

Table 5.3-4 True table for the detection logic

Previous Result	Current Result	Increment En	Clear
1	1	1	0
1	0	0	1
0	1	1	0
0	0	0	0

✧ Parallelization of Huffman Coding

The Huffman coding stage replaces all the non-zero coefficients in the 8×8 block with a pair of variable-length codewords. One codeword contains the value of the coefficients. Another codeword contains the bit-length information of the first codeword. Both codewords can be obtained by looking up into certain Huffman tables using the coefficient value as the index or address.

One possible approach, which requires no hardware change, is shown in **Figure 5.3-3**.

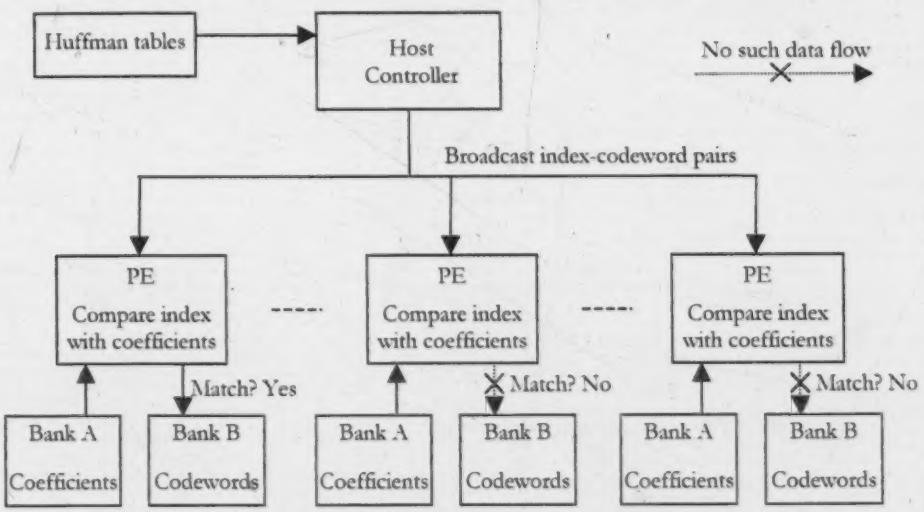

Figure 5.3-3 Parallelized Huffman coding on the current DSP-RAM architecture

In this approach, the Huffman tables are broadcast to the PEs in the form of index-codeword pairs. By comparing the index with the coefficients in the local

memory, each PE will determine if the codeword will be stored into its memory to represent the coefficients. For example, for a codeword 0111 that can be found in a Huffman table with an index 45, the index 45 and the codeword 0111 will be broadcast to PEs in two clock cycles. All the PEs will scan the coefficients simultaneously. If they find a coefficient with a value of 45, then the coefficient will be represented with the codeword. If the coefficient is not 45, then the memory write will be disabled and no value is written.

Although this approach requires no hardware change, it is not efficient in terms of performance. For every entry of a Huffman table, three clock cycles are generally needed to determine if the codeword from the Huffman table should be written to replace the coefficient value in memory, i.e. two clock cycles for loading the index and the coefficient values, and one clock cycle for comparing and writing. For 8-bit coefficients, the size of the Huffman tables should be 255 (256-1, the value 0 is not considered in Huffman coding stage). Therefore, approximately 765 (255×3) clock cycles are needed for a PE to encode one coefficient. Similarly, for 16-bit coefficients, approximately 196,605 (65535×3) clock cycles are needed per coefficient. In contrary, if the effective bit-length of coefficients is 4 on average, only approximately 12 instructions (estimated from the IJG source codes) are needed for the IJG Huffman encoder to encode one coefficient.

If we add a bank of content-addressable memories (CAM) of size 1 Kbytes into DSP-RAM, as shown in **Figure 5.3-4**, a PE can directly use the values of the coefficients in the bank A as addresses to load the codewords from bank C and store them in bank B. Using this approach, it's possible for a PE to encode one coefficient in about 6 clock cycles.

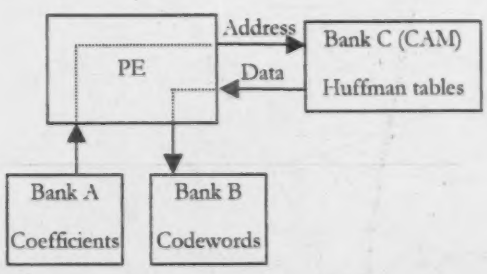

Figure 5.3-4 Using CAM to simplify the table-lookup operation

Obviously, the second approach outperforms the first approach. However, it increases the transistor budget significantly. An alternative approach would be to enable or reconfigure one of its memory banks to work as a CAM. This could be realized by adding an address register into the DSP-RAM architecture as shown in **Figure 5.3-5**. The address register can be sourced from either of the two memory data buses and can be used to generate addresses for either of the two memory banks. Since each PE can now generate different addresses for itself, it can access memory independently of the other PEs.

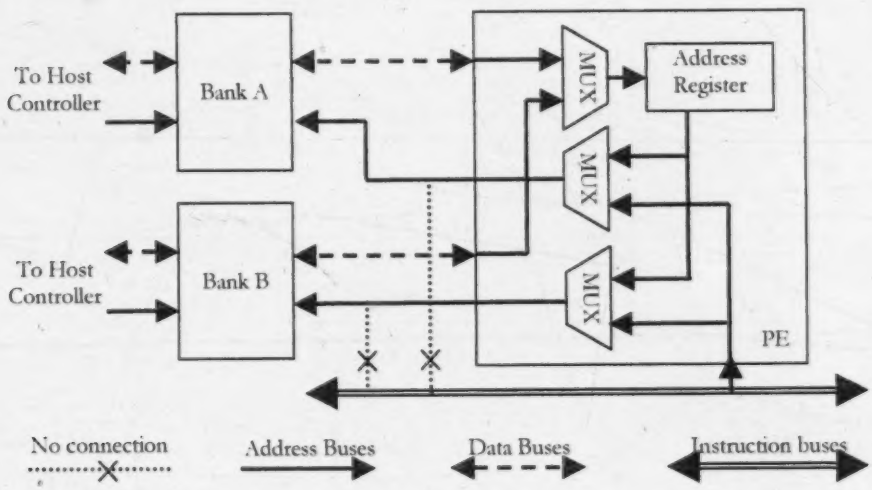

Figure 5.3-5 Adding an address register to realize table lookup in DSP-RAM

In the third approach, a coefficient could also be encoded in about 6 clock cycles for one PE. Since the other stages of JPEG encoding require both memory banks for the processing, we can not store Huffman tables permanently in one of the memory banks. Therefore, we have to load Huffman tables every time when we need to process Huffman encoding on a new group of MCUs. This adds an overhead of 1024 clock cycles (for 8-bit coefficients) to every run of the process. The more PEs that we have on one chip, the more this overhead can be averaged by more coefficients. Suppose we can still integrate 32 PEs on one chip, every PE process six 8×8 blocks of coefficients, then the average coding speed is about 0.271 clock cycles per coefficient (6/32 + 1024/(32×6×64)). If DSP-RAM is operated at 200 MHz, while the IJG Huffman encoder is executed on a 1.6-GHz Pentium 4 computer, the DSP-RAM

implementation of the Huffman encoder is approximately 4.9 times faster than the IJG Huffman encoder.

5.3.4. Upgrade to Motion JEPG Encoder

Motion JPEG, commonly known as MJPEG, compresses each frame of a video with JPEG compression technology. If we use the DSP-RAM JPEG encoder to compress each frame sequentially, then the DSP-RAM JPEG encoder can be upgraded to become a DSP-RAM MJPEG encoder.

The DSP-RAM JPEG encoder uses the MCU as the basic unit of processing. Each PE process one MCU a time. Depending on how many PEs the DSP-RAM has, we can compress a group of MCUs on the DSP-RAM at the same time. When all of the MCUs of a frame are compressed, the MCUs from the next frame will be input into the DSP-RAM. The host controller will handle the sequential parts of the processing and synchronize the processing of the frames. **Figure 5.3-6** shows the data flow of such a process.

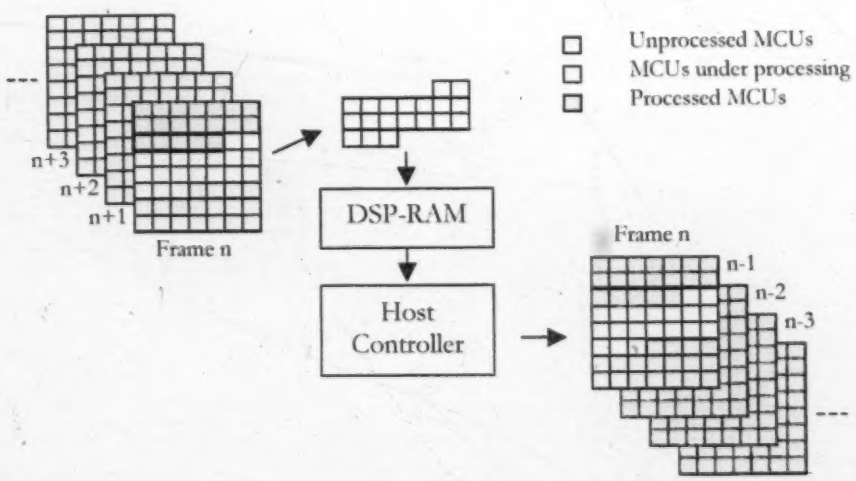

Figure 5.3-6 Data flow of a proposed DSP-RAM MJPEG encoder

Chapter 6

CONCLUSIONS

6.1 Summary

As an integrated memory-processor SIMD architecture, DSP-RAM has advantages in terms of performance, power consumption and scalability for functions and algorithms with intensive multiply-accumulate vector computations. However, to fully deliver on its potential, DSP-RAM has to be able to deal efficiently with the entire system-level problem as well as kernel sub-problems.

In our research, we investigated and improved DSP-RAM's ability to deal with system-level problems. The main problem we used to challenge DSP-RAM is JPEG image compression. First, we thoroughly analyzed DSP-RAM's versatility and found out that DSP-RAM is efficient at dealing with regular vector operations while being awkward at performing complex scalar operations. Hence we performed the regular vector operations to be performed on DSP-RAM and kept the complex scalar operations onto the host controller. Second, we investigated the structure and behavior of the PE pipeline, analyzed how hazards are generated, and determined how hazards affected the overall performance of the DSP-RAM implementation. Based on the analysis, we summarized the efficient methods to avoid pipeline hazards and to reduce the overhead brought by data organization and initialization. In this way, we better utilized DSP-RAM's computational power and improved the efficiency of the DSP-RAM implementation. Then, in order to improve DSP-RAM's programmability and to reduce the complexity of controlling DSP-RAM's pipeline, we developed a programming model that utilized three state machines to control the input, output and the operation of the PE pipelines.

Based on the analysis of DSP-RAM's ability to deal with system-level problems, we successfully implemented a parallel JPEG encoder and a parallel DWT (one wavelet stage only) on DSP-RAM. The parallel algorithms output the same results as their sequential counterparts.

To obtain the hardware resource requirements and the maximum operating clock frequency of DSP-RAM, we optimized and implemented DSP-RAM architecture on the latest FPGA platform from Xilinx. The resulting FPGA implementation of DSP-RAM contains 32 PEs and 64 memory blocks, and works at a clock frequency of about 226 MHz.

Our evaluation of the parallel JPEG encoder shows that, with the assistance of a DSP-RAM chip (32 PE @ 200 MHz), a host controller that is 3.5 times slower than a Pentium 4 @ 1.6 GHz microprocessor, could deliver comparable computational power for the JPEG encoding problem as the Pentium 4 microprocessor. Even when the Pentium 4 uses the SSE2 SIMD extension, the DSP-RAM enhancement can work at a clock frequency 8 times slower while still delivering slightly better performance.

There are two factors that might affect the accuracy of our results. First, the assumption that DSP-RAM can run at a clock speed of 200 MHz level was made based on the simulation results of the FPGA implementation of DSP-RAM. However, our simulations ignored the effects of those off-chip signals. It's possible that the loads on those signal lines and pins might slow down the maximum clock speed of the DSP-RAM. Second, it is possible to utilize the computational power of the host controller and allow it to process in parallel with DSP-RAM. By ignoring the last possibility, our simulation results would tend to be more conservative.

Our analysis also shows that, to further improve DSP-RAM's computational power, it would be more efficient to increase the functional versatility of PE rather than to further increase the number of PEs or the PE clock frequency. We also proposed and investigated possible approaches to parallelize entropy encoding on the DSP-RAM architecture.

6.2 Future work

An immediate goal for further work on DSP-RAM JPEG encoder is to fully implement entropy encoding on DSP-RAM. Two basic operations of entropy encoding are table look-up and word-length calculation. Since these two operations are also widely used in other DSP algorithms, the versatility of DSP-RAM would be enhanced greatly if we could add these functions.

An important research topic concerning DSP-RAM is the interface between DSP-RAM and its host controller. A standard interface would make DSP-RAM easier to use and would be critical for pushing DSP-RAM into markets. Regarding the application of DSP-RAM architecture to other areas, an interesting idea is to develop DSP-RAM as a stand-alone high performance reconfigurable firmware, which works very much like an FPGA. The hardware could be designed as a generic platform with standard interface so that it could be integrated into embedded system easily. The embedded software could be designed like BIOS or the programmable stream files of FPGAs, which determines the functionality of DSP-RAM. Users could program DSP-RAM by themselves, or utilize off-the-shelf IP blocks, which would hide the implementation details of the function and would be optimized to deliver high computational power.

BIBLIOGRAPHY

1. D.G. Elliott, "Computational RAM: A Memory-SIMD Hybrid," Ph.D. thesis, Dept. of Electrical Engineering, University of Toronto, Canada, 1998.
2. D.G. Elliott et al., "Computational RAM: Implementing Processors in Memory," *IEEE Design & Test of Computers*, vol. 16, no. 1, Jan.-Mar., 1999, pp. 32-41.
3. H. Ai, N. Li, T. Li, M. K. Mandal, and B. F. Cockburn, "Efficient Implementation of Motion Estimation Technique on the Computational RAM Architecture," *2002 IEEE Canadian Conference on Electrical and Computer Engineering*, Winnipeg, MB, Canada, May 12-15, 2002, pp. 609-613.
4. T. Le et al., "Computational RAM Implementation of Mean-Average Scalable Vector Quantization for Real-Time Progressive Image Transmission," *1996 IEEE Canadian Conf. Electrical and Comp. Eng.*, Calgary, AB, May 1996, pp. 442-445.
5. D. G. Elliott, W. M. Snelgrove, and M. Stumm. "Computational RAM: A Memory-SIMD Hybrid and its Application to DSP," *Custom Integrated Circuits Conference*, Boston, MA, May 1992, pp. 30.6.1--30.6.4.
6. Z. Wang,i B. F. Cockburn, D. G. Elliott, and W. Krzymien, "DSP-RAM: A Logic-Enhanced Memory Architecture for Communication Signal Processing," *1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, Aug. 22-24, 1999, pp. 475-478.
7. B. S.-H. Kwan, B. F. Cockburn, and D. G. Elliott, "Implementation of DSP-RAM: An Architecture for Parallel Digital Signal Processing," *2001 IEEE Canadian Conference on Electrical and Computer Engineering*, Toronto, ON, Canada, May 20-23, 2001, pp. 341-346.
8. B. S.-H. Kwan, "Implementation and Evaluation of DSP-RAM: An Architecture for Parallel Digital Signal Processing," M.Sc. thesis, Dept. of Electrical Engineering, University of Alberta, Canada, 2001.
9. S. J. Dillen and B. F. Cockburn, "Parallel Filtering and Thresholding of images on the SIMD DSP-RAM Architecture," *2002 IEEE Canadian Conference on Electrical and Computer Engineering*, Winnipeg, MB, Canada, May 12-15, 2002, pp. 995-1000.

10. D. Burger, J. Goodman, "Billion-transistor architectures", *IEEE Computer*, Sept. 1997, pp. 46-47.
11. M. Lipasti and J. Shen, "Superspeculative microarchitecture for beyond ad 2000", *IEEE Computer*, September 1997, pp. 59-66.
12. S. Eggers, et al. "Simultaneous Multithreading: A Platform for Next-generation Processors", *IEEE Micro*, September/October 1997, pp. 12-18.
13. L. Hammond, B. A. Nayeh and K. Olukotun, "A single-chip multiprocessor", *IEEE Computer*, September 1997, pp. 79-85.
14. A. Agarwal, et al. "Baring it all to Software: Raw Machines", *IEEE Computer*, September 1997, Pages 86-93.
15. K. Diefendorff and P. K. Dubey, "How Multimedia Workloads will Change Processor Design," *IEEE Computer*, vol. 30, no. 9, pp. 43--45, Sept. 1997.
16. Christoforos E. Kozyrakis et al. "Scalable Processors in the Billion-Transistor Era: IRAM", *IEEE Computer*, September 1997, pp. 75-78.
17. C. Kozyrakis, et al. "Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM", *12th Hot Chips Conference*, Palo Alto, CA, August 2000.
18. C. Kozyrakis, D. Patterson, "Overcoming the Limitations of Conventional Vector Processors", *30th International Symposium on Computer Architecture*, San Diego, CA, June 2003.
19. Z. Wang, et al. "DSP-RAM: A Logic-Enhanced Memory Architecture for Communication Signal Processing", *1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, Aug. 22-24, 1999, pp. 475-478.
20. C. Kozyrakis and D.A. Patterson, "A New Direction for Computer Architecture Research," *IEEE Computer*, November 1998, pp. 24-32.
21. T.M. Conte, et al. "Challenges of Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Vol. 30, No. 12, Dec. 1997, pp. 33-37.
22. D.A.Patterson, J.L.Hennessy, "Computer architecture: A quantitative approach", Second Edition, *Morgan Kaufmann Publishers Inc.*, San Francisco,1996.
23. White paper: "Hyper-Threading Technology on the Intel Xeon Processor Family for Servers", *Intel Corporation*, 2002.

24. White paper: "UltraSPARC® IIIi Architecture Overview", Version 1.0, *Sun Microsystems*, April 2003.
25. White paper: "MPC750 RISC Microprocessor Technical Summary", *Motorola Inc.* 1997.
26. V. Bhaskaran and K. Konstantinides. "Image and Video Compression Standards, Algorithms and Architectures," Second Edition, *Kluwer Academic Publishers*, 1999.
27. JPEG official site, http://www.jpeg.org/.
28. DSPworx, Inc. A webpage about the features of JPEG 2000, http://www.dspworx.com/primer_jpeg2000.htm, 2002.
29. Image power, Inc. "JPEG 2000: Comparison between JPEG and Image Power JPEG 2000", http://www.imagepower.com/technology/jpeg2000/compare/.
30. Eric Hamilton, "JPEG File Interchange Format", Version 1.02, September 1, 1992.
31. Charles Poynton, "A guided Tour of Color Space", *New Foundations for Video Technology, Proceedings of the SMPTE Advanced Television and Electronic Imaging Conference*, San Francisco, Feb. 1995, pp. 167-180.
32. Independent JPEG Group, Source codes for JPEG Image Compression Library, Release 6b, Mar. 27, 1998.
33. Adobe Developers Association, "TIFF Specification", Revision 6.0, *Adobe Systems, Inc.*, June 3, 1992.
34. C. Loeffler, A. Ligtenberg and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", *Proc. Int'l. Conf. on Acoustics, Speech, and Signal Processing 1989 (ICASSP '89)*, pp. 988-991.
35. Subhasis Saha, "Image Compression – from DCT to Wavelets: A Review", http://www.acm.org/crossroads/xrds6-3/sahaimgcoding.html, *ACM Crossroads Student Magazine*, Jan. 21, 2001.
36. R. M. Russell, "The CRAY-1 computer system", *Communications of the ACM*, 21(1):63-72, 1978.
37. J. S. Kolodzey, "CRAY-1 computer technology", IEEE Transactions on Components, Hybrids, and Manufacturing Technology, CHMT-4(2), pp. 181-187, June 1981.

38. JPEG, "JPEG 2000 Image Coding System", *JPEG 2000 Final Committee Draft Version 1.0*, March 16, 2000.

39. Xilinx Inc., "Virtex-II Pro Platform FPGAs: Advance Product Specification", *DS083-1 (v2.4.1)*, March 24, 2003.

40. Intel Corporation, "Integrated Performance Primitives for Intel Architecture: IJG Library Modification by Using IPP", Version 2.0.1, November 2001.